An Interactive Theorem Prover for First-Order Dynamic Logic

Tuhin Kanti Das

Department of Computer Science

Supervisor

Professor Michael Winter

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

© Tuhin Kanti Das, 2012

To Professor Michael Winter
&
My Parents

Abstract

Dynamic logic is an extension of modal logic originally intended for reasoning about computer programs. The method of proving correctness of properties of a computer program using the well-known Hoare Logic can be implemented by utilizing the robustness of dynamic logic. For a very broad range of languages and applications in program verification, a theorem prover named KIV (Karlsruhe Interactive Verifier) Theorem Prover has already been developed. But a high degree of automation and its complexity make it difficult to use it for educational purposes. My research work is motivated towards the design and implementation of a similar interactive theorem prover with educational use as its main design criteria. As the key purpose of this system is to serve as an educational tool, it is a self-explanatory system that explains every step of creating a derivation, i.e., proving a theorem. This deductive system is implemented in the platform-independent programming language Java. In addition, a very popular combination of a lexical analyzer generator, JFlex, and the parser generator BYacc/J for parsing formulas and programs has been used.

Acknowledgements

First and foremost, I offer my sincerest gratitude to my supervisor, Professor Michael Winter, for his guidance, caring, patience, and providing me with an excellent atmosphere for doing research. Without his enthusiasms and support from the initial to the final stage of the development of this dissertation, the completion of this thesis would never have been possible. I must have to admit that, one simply could not wish for a better supervisor. I attribute the level of my Masters degree to his encouragement and effort.

I would like to express sincere appreciation to the committee members for their invaluable guidance and encouraging me with many insightful thoughts.

I would like to thank the department of Computer Science and the Graduate Studies of Brock University for the financial and academic support. It was not possible to pursue my research in an anxiety-free environment without the support I was provided with.

I would also like to thank my family members and friends for their constant assistance, and inspiration with the best wishes. They were always there cheering me up, and stood by me through the good times and bad.

T.K.D

Contents

1 Introduction **1**

2 Background on Dynamic Logic **3**

2.1 Modal Logic 3

2.2 Dynamic Logic 4

2.2.1 Intuitive meaning of two modal operators of DL 4

2.2.2 Relationship of DL with Hoare logic 5

2.3 First-order dynamic logic 9

2.3.1 Syntax 9

2.3.2 Semantics 13

2.3.3 Satisfiability and Validity of Formulas 17

2.4 Reasoning in FODL 18

3 Calculus of the Deductive System **20**

3.1 Inference Rules 20

3.2 Soundness of the inference rules 28

3.3 Incompleteness of this system 33

4 Overview of the System **35**

4.1 Demonstration of the system 35

4.2 Demonstration of the inference rules 39

5 Implementation **50**

5.1 Brief explanation of source code 50

5.2 Class hierarchies 52

6 User Manual **55**

7 Conclusion **64**
7.1 Summarization of work 64
7.2 Comparison to other systems 65
7.3 Future work 66

List of Tables

2.1 List of well-known modal logic 3

3.1 Induction Rule 23
3.2 acap 23
3.3 Logical rules for first-order reasoning 25
3.4 Inference rules for dynamic reasoning 26

List of Figures

2.1 Comparison of values of i and r in the given example 8

4.1 Example of Valid and Invalid formulas 36
4.2 A valid formula is entered 36
4.3 After application of the implication rule 36
4.4 User is asked to enter the loop invariant 37
4.5 While rule has been applied 37
4.6 List of axioms is loaded 38
4.7 Application of user selected axiom 38
4.8 A portion of the proof is done 38
4.9 A completed proof 39
4.10 Application of the AndLeft rule 40
4.11 Application of the AndRight rule 40
4.12 Application of the OrLeft rule 41
4.13 Application of the OrRight rule 41
4.14 Application of the ImpLeft rule 42
4.15 Application of the ImpRight rule 42
4.16 Application of the Induction rule 43
4.17 Application of the NotLeft rule 43
4.18 Application of the NotRight rule 44
4.19 Application of the PBC rule 44
4.20 Application of the Assignment rule 45
4.21 Application of the if-else rule 45
4.22 Application of the loop rule 46
4.23 Application of the ForallLeft rule 47
4.24 Application of the ForallRight rule 48
4.25 Application of the ExistsLeft rule 48
4.26 Application of the ExistsRight rule 49

5.1 Class diagram of Formula 53
5.2 Relationship between Formula and Term 53
5.3 Class diagram of Program 54
5.4 Class diagram of Term 54
5.5 Relationship between Formula and Program 54

6.1 The user interface of this system 56
6.2 The current goal is displayed in the `Assertions` window 56
6.3 The assumption of the current proof is displayed in the `Assumptions` window 57
6.4 Activation of a left-hand rule, AndLeft 57
6.5 The left-hand rule, AndLeft, has been applied 58
6.6 Selection of formula in `Assertions` window 59
6.7 Application of derivation button in `Assertions` window 59
6.8 Application of an axiom rule 60
6.9 Mapping of variable a to i in unifier 60
6.10 Mapping of variable a to i in unifier 61
6.11 Application of unification 62
6.12 The completion of the main proof 62
6.13 Maximized view of `Proof Explorer` window 63

Chapter 1

Introduction

Dynamic Logic is a formal system for reasoning about programs. It can be described as a mixture of three complementary classical ingredients: first-order predicate logic, modal logic, and the algebra of regular events. These make it rich in both theoretical and practical aspects [5].

Dynamic Logic and other program logics are intended to facilitate the process of producing programs correctly. For the application of formal verification tools, we need to have a formal specification of correctness for the verification tools to work with. In general, we can control the behavior of a program by a formal description of correctness specification. That means, if the behavior of a given program satisfies a correctness specification, then this program is correct with respect to that specification.

The effectiveness of dynamic logic for program verification motivated several existing program verification tools implementing various variations of calculi based on dynamic logic. Example systems include KIV (Karlsruhe Interactive Verifier) and KeY system [4, 21]. KIV is a formal system development tool which supports proofs for specification validation, design verification, and program verification using an advanced interactive deduction component based on proof tactics. It is a very powerful tool for program verification which can be used for professional purposes. On the other hand, the KeY System is a formal software development tool for the verification of object-oriented software based on first order dynamic logic. But a high degree of automation and complexity of use make these existing systems very difficult to be employed as educational tools.

This thesis is about the implementation of a software verification tool for educational purposes. It is targeted for the students learning software verification techniques. Key features of this system are its ease of use, and the visual exploration of every step of program verification. The main requirement of this system is to define and implement a formal calculus for reasoning about programs. Rules of Hoare logic have been implemented in this system to fulfill this requirement. In addition to that, we were also required to implement the rules of classical first order logic. But those rules were already implemented for relational algebras by Bahar Aameri in her MSc Thesis [1]. So, we have reused her code with some modification to cope with the requirements of our system.

Chapter 2

Background on Dynamic Logic

2.1 Modal Logic

Modal logic is a type of formal logic that extends classical propositional and predicate logic to include operators expressing modality. With the intention of distinguishing two sorts of truth: necessary truth and mere contingent truth, it was originally invented by Lewis (1918). But the term `modal logic` is used more broadly to cover a whole class of different logics with similar rules and a variety of different symbols. A list of well known examples of modal logic is given in Table 2.1 [11, 12]:

Logic	Symbol	Expressions Symbolized
Modal Logic	$\Box$	It is necessary that ...
	$\Diamond$	It is possible that ...
Deontic Logic	O	It is obligatory that ...
	P	It is permitted that ...
	F	It is forbidden that ...
Temporal Logic	G	It will always be the case that ...
	F	It will be the case that ...
	H	It has always been the case that ...
	P	It was the case that ...
Doxastic Logic	Bx	x believes that ...

Table 2.1: List of well-known modal logic

2.2 Dynamic Logic

Dynamic logic (DL) is a modal logic which is originally intended for reasoning about computer programs. In this logic, we have countably many $\Box$ and $\Diamond$ operators. In fact, for each program there exists a $\Box$ and a $\Diamond$ operator [13].

2.2.1 Intuitive meaning of two modal operators of DL

If φ is a formula of DL and α is a program, then the meaning of the two modal operators can be intuitively described as:

- **$[\alpha]\varphi$:** every computation of α that terminates leads to a state satisfying φ.
- **$\langle\alpha\rangle\varphi$:** there is a computation of α that terminates in a state satisfying φ.

In the above definitions, $\langle\alpha\rangle\varphi$ asserts the total correctness of a program. In general, a program is designed for implementing some functionalities which can be expressed formally in the form of an input/output specification. Such a specification involves an input condition or precondition φ and an output condition or postcondition ψ. These two conditions are properties of input state and output state respectively. The program is supposed to terminate in a state where the output condition is true given the input state satisfies the input condition. Now, the program is called partially correct with respect to the input/output specification φ, ψ if the following condition is true:

- Whenever the program is started in a state where the input condition φ is true, then if the program ever terminates, it does so in a state where the output condition ψ is true.

So, the definition of partial correctness does not ensure the termination of the program. In contrast, we can call a program totally correct with respect to an input/output specification φ, ψ if the following two conditions are satisfied [14, 15, 16, 17]:

- The program is partially correct with respect to the specification.
- The program terminates whenever it starts in a state satisfying φ.

2.2.2 Relationship of DL with Hoare logic

Hoare logic is a calculus that can be used to prove partial correctness assertions, so called Hoare-triples, which take the form [3]:

$$\{\varphi\}\mathrm{c}\{\psi\},$$

where φ, ψ are formulas and c is a command (or program). It states that if the pre-condition φ is true before the execution of the program c and the program terminates, then the post-condition ψ is satisfied after the program execution. For example, let us consider the Hoare-triple, $\{n \geq 0\}$ p $\{r = n^2\}$, where p is a program. In this example, if the precondition $n \geq 0$ is true before the execution of the program p, and if p terminates, then we will be ended up in a state where the postcondition $r = n^2$ is satisfied.

Six rules of Hoare logic that are used for program property verification are given below:

- **Skip:** $\{\varphi\}$ skip $\{\varphi\}$
- **Assignment:** $\{\psi[a/x]\}$ x:=a $\{\psi\}$, where $\psi[a/x]$ denotes the substitution of a for x.
- **Sequencing:** $\frac{\{\varphi\}C_0\{\chi\} \quad \{\chi\}C_1\{\psi\}}{\{\varphi\}C_0;C_1\{\psi\}}$
- **Conditional:** $\frac{\{\varphi\wedge b\}C_0\{\psi\} \quad \{\varphi\wedge\neg b\}C_1\{\psi\}}{\{\varphi\}if\, b\, then\, C_0\, else\, C_1\, fi\{\psi\}}$
- **Loop:** $\frac{\{\varphi\wedge b\}C\{\varphi\}}{\{\varphi\}while\, b\, do\, C\, od\{\varphi\wedge\neg b\}}$
- **Consequence:** $\frac{\varphi\rightarrow\varphi' \quad \{\varphi'\}C\{\psi'\} \quad \psi'\rightarrow\psi}{\{\varphi\}C\{\psi\}}$

Rules are written with premises above a horizontal line, and the conclusions below it. If some conditions need to be satisfied for the application of the rule, it is written by its side. In a formal proof system, a collection of rules is used to carry out stepwise deductions. If the number of premises in a rule is zero, it is called an axiom. A derivation using these rules generates a proof tree which consists of nodes that represents the instances of applied rules. Leaves of a proof tree represent either the instances of axioms of the system or unfinished sequents whose derivations are not finished yet. Therefore, a derivation tree for a derivable sequent will not have any unfinished leaves and such a derivation tree can be called finished. For a derived rule, a derivation tree will have the premises of the rule as its unfinished leaf sequents.

For the purpose of dynamic reasoning we have implemented the rules of Hoare logic in a more compact form. But these implemented rules are equivalent to the corresponding rules in Hoare logic. We had to implement three rules of Hoare logic namely, assignment rule, conditional rule and loop rule. The skip rule is trivial and the rest of the rules, namely, sequencing rule and consequence rule are implicitly there. We will introduce the implemented version of these rules in detail in the following chapters.

Let us give an example of program property verification using Hoare logic. In this example, we have formulas n≥0 and r=n*m as the pre-condition and the post-condition of the program, respectively. The verification method is started here from the post-condition, and the rules of Hoare logic are applied backwards. Instead of using the tree form of a derivation, here we annotated each of the program statements with a pre-condtion and a post-condition. In this example, as the last statement of the program is a while statement, the loop rule of Hoare logic is applied first. Here, each program statement is correct with respect to the corresponding pre-condition and post-condition, and the correctness of each statement is validated by the application of a specific Hoare logic rule. For example, if we consider the first statement of the

{n≥0}

↓

{I(0, 0)} r:=0; {I(r, 0)}
{I(r, 0)} i:=0; {I(r, i)}
{I(r, i)} while i < n do
{I(r, i)} ∧ {$(i < n)$}

↓

{I(r+m, i+1)} r:=r+m; {I(r, i+1)}
{I(r, i+1)} i:=i+1; {I(r, i)}
od $\{I(r, i)\} \wedge \neg\{(i < n)\}$

↓

{r=n*m}

Figure: Verification of program property using Hoare logic

program, r:=0, we have I(r, 0) as the post-condition. Notice that, I(r,0) is a placeholder for a formula, an instance of the so-called loop invariant. Application of the assignment rule to this program statement towards backword direction will generate the pre-condtion I(0, 0). So, we can conclude that, this statement is correct with re-

spect to the pre-condition, I(0, 0) and the post-condition, I(r, 0), and its correctness is validated by the application of the assignment rule of Hoare logic. In addition, if we move from one statement to another, the application of the sequencing rule of Hoare logic is implicitly there. Suppose, let us consider the first two assignment statements of the program in the given example. We have used the pre-condition of the second assignment statement, i:=0, as the post-condition of first assignment statement r:=0. It is possible to use in this way because of the implicit application of sequencing rule of Hoare logic. The downwards arrow symbols in the given example represents the logical operation of implication, and the application of the consequence rule of Hoare logic is embedded here. For example, if we consider the first and the last occurrences of the arrow symbol in this example, first occurrence represents the formula,

$$n \geq 0 \rightarrow I(0, 0),$$

and the last one represents the implication,

$$I(r, i) \land \neg(i < n) \rightarrow r = n*m.$$

If we consider the whole program as a symbol C, we can write,

$$\{I(0, 0)\}\ C\ \{I(r, i) \land \neg(i < n)\}.$$

An additional proof of $I(r, i) \land \neg(i < n) \rightarrow r = n*m$, and the implicit application of the consequence rule of Hoare logic gives,

$$\{n \geq 0\}\ C\ \{r = n*m\}.$$

As mentioned above, the notation I(r, i) is the placeholder for an invariant of the program. A loop invariant is used for proving properties of a loop. This is a first-order formula that should be true at the beginning of each loop iteration and is guaranteed to remain true on every iteration of the loop. Therefore, on the exit point of the loop, both the loop invariant and the loop termination condition can be guaranteed. It is not an easy task to find a sufficiently strong loop invariant for verifying a program property. In addition to interactions on the level of first-order reasoning, human interaction is required to handle loops in a program. We will prompt the users of our system to enter a suitable loop invariant interactively for proving program properties in this deductive system [10, 18].

As we mentioned earlier, we started the verification procedure from the bottom of the program. Following the loop rule of Hoare logic, we placed the formula $I(r, i) \land \neg(i < n)$ after the exit point of while loop and the formula I(r, i) before the entry point of while loop. Now, as the assumption of the loop rule, that is the formula we have above the bar suggests, we placed I(r, i) before the exit point of while loop and $I(r, i) \land (i < n)$ after the entry point of while loop. Now we work backwords by applying the

assignment rule of Hoare calculus. As we move backwards, the sequencing rule and the consequence rule of Hoare logic are implicitly there, and incorporation of these rules let us to move from down to top of the program.

The verification of properties of the above program requires proof of the following three steps:

- $n \geq 0 \rightarrow I(0, 0)$
- $I(r, i) \wedge (i < n) \rightarrow I(r+m, i+1)$
- $I(r, i) \wedge \neg(i < n) \rightarrow r = n*m$

i (=i+1)	r (= r+m)
0	0
1	0+m=1*m
2	m+m=2*m
3	2*m+m=3*m
. . .	. . .
n	n*m

Figure 2.1: Comparison of values of i and r in the given example

Now, for verifying program property using Hoare logic, we need to determine the invariant of the program. An invariant of a program is a first-order formula which is valid at the beginning of each loop iteration [10]. By comparing the values of i and r we can determine the invariant of the above program, which can be given as, $I(r, i) \equiv (r = i*m) \wedge (i \leq n)$. Using this invariant, proofs of the three steps are illustrated below.

Proof of 1st step:

- By applying the arithmetic axiom, $n \geq 0 \Leftrightarrow 0 \leq n$, we can get,

$$n \geq 0 \Leftrightarrow (0 = 0*0) \wedge (0 \leq n)$$
$$\Rightarrow I(0, 0) \qquad \square$$

Proof of 2nd step:

- $I(r, i) \wedge (i < n) \equiv (r{=}i*m) \wedge (i \leq n) \wedge (i < n)$
  $\Rightarrow (r{=}i*m) \wedge (i < n)$
  $\Rightarrow r+m = i*m+m \wedge i+1 \leq n$
  $\Rightarrow r+m=(i+1)*m \wedge i+1 \leq n$
  $\Rightarrow I(r+m, i+1)$ □

Proof of 3rd step:

- $I(r, i) \wedge \neg(i < n) \equiv r{=}i*m \wedge (i \leq n) \wedge (i \geq n)$
  $\Rightarrow r{=}i*m \wedge i{=}n$
  $\Rightarrow r{=}n*m$ □

We can see DL as a generalization of Hoare logic. In terms of expressiveness, dynamic logic is more powerful than Hoare logic because, in addition to partial correctness, program termination and program implication can also be expressed in DL. Let us consider the following DL formula [4, 10, 15]:

$$\varphi \Rightarrow [\alpha]\psi,$$

where φ and ψ are first-order formulas. This formula has the same meaning as the Hoare triple, $\varphi\{\alpha\}\psi$: given α is executed in a state satisfying φ, if the program α terminates, the final state will satisfy ψ.

2.3 First-order dynamic logic

In this section, we are going to explore a specific version of dynamic logic, namely, first-order dynamic logic. This version of dynamic logic will be used as the logical basis of our deductive system.

2.3.1 Syntax

The language of first-order Dynamic Logic is built upon classical first-order logic. A set of programs and a set of formulas are defined on top of a vocabulary of function symbols and predicate symbols $\sum$. These two sets use the modal construct [ ] for interacting with each other. Programs and formulas in DL are usually defined by mutual recursion.

Let us consider a finite set of vocabulary $\sum = \{$ f, g,..., p, r,...$\}$, where, f, g and p, r denote typical function symbols and predicate symbols of the vocabulary $\sum$. Each of these function and predicate symbols is implicitly associated with a fixed

arity, the number of arguments it has. Functions and predicates of arity 0, 1, 2, 3 and n are called nullary, unary, binary, ternary, and n-ary, respectively. We will use a countable set of individual variables V={x_0, x_1, ...}. Nullary function symbols are called constants and nullary predicate symbols are called propositions of the language.

Definition 1. *The set Term of terms is defined recursively by the following:*

- *Each variable $x \in V$ is a term, i.e., $V \subseteq$ Term.*
- *If $f \in F$ is an n-ary function symbol and t_1, t_2, ... , $t_n \in$ Term are terms, then $f(t_1, t_2, ..., t_n) \in$ Term.*

Examples of terms include f(x, y, z) or f(g(x,x), g(x,y), y); where f is a ternary function symbol and g is a binary function symbol. In this system, we have defined arithmetic operators, PLUS(+), MINUS(-), MULTIPLICATION(*), DIVISION(/) and EXPONENT(ˆ) as the function symbols. In addition, here we have implemented the INFIX notation of writing arithmetic expressions, which dictates us to write expressions in the form of x+y, instead of +(x, y), or (x, y)+. So, a term in this system can take the form of a+b, a-b, a*b, a/b or aˆb, where a ∈ Term, b ∈ Term.

Atomic Formulas and Programs

In all versions of DL, formulas that take the form, $r(t_1, t_2, ..., t_n)$, where r is an n-ary predicate symbol of $\sum$ and t_1, t_2, ... , t_n are terms of $\sum$, are called atomic formulas of the first-order vocabulary $\sum$. In this deductive system, a predicate symbol can be any of the following constructs: =, <, >, ≤, ≥. So example of an atomic formula can take the form of a ≤ b or a+b ≥ c-d, where {a, b, c, d} ⊆ Term.

Various programming constructs are used for defining programs inductively from atomic programs. Selection of different classes of atomic programs and programming constructs will result in different classes of programs. Giving meaning to a compound program requires to give meanings to its constituent parts. If $x \in \sum$ and t is a term of $\sum$, an atomic program can be defined as a simple assignment, $x := t$.

Tests

Originally, in DL, conversion of a formula into a program requires a test operator ?. For our version of implementation, we will allow only quantifier-free first-order formulas as tests. This version of test is called poor test. Another version of tests,

called as rich test, does not place any restrictions on the form of tests and allows any DL formulas. So, using rich tests, a formula can contain other programs, which could be other tests, which makes it impractical to implement. In the poor test version, we can define programs independently from formulas. In this paper we will use the term test to indicate the poor test version. We have implemented the poor test version as part of if-else and while program, but we have skipped the notation ? in the implementation. So the test in our implementation is not a standalone program, it is used only as the condition of an if-else or a while statement.

Regular Programs

For the above mentioned set of atomic programs and tests, the set of regular programs in this system is defined as:

- Any atomic program is a program, i.e., an assignment statement is a program.
- If α and β are programs, then $\alpha;\beta$ is a program.
- If φ is a quantifier-free first-order formula, and α and β are programs, then

  $$\text{if}(\varphi) \text{ then } \alpha \text{ else } \beta \text{ fi},$$

  is also a program.
- If φ is a quantifier-free first-order formula, and α is a program, then

  $$\text{while}(\varphi) \text{ do } \alpha \text{ od}$$

  is also a program.

As we mentioned earlier, the formula φ occuring in the above programs must be a quantifier-free first-order formula.

Formulas

In FODL, a formula is defined similarly to a formula of first-order logic, with the addition of a rule for modality. In our deductive system a formula is defined in the following way:

- The false formula is a formula.
- The true formula is a formula.
- Any atomic formula is a formula.
- If φ is a formula, then $\neg\varphi$ is a formula.

- If φ and ψ are formulas, then $\varphi \wedge \psi$ is a formula.
- If φ and ψ are formulas, then $\varphi \vee \psi$ is a formula.
- If φ and ψ are formulas, then $\varphi \rightarrow \psi$ is a formula.
- If φ and ψ are formulas, then $\varphi \leftrightarrow \psi$ is a formula.
- If φ is a formula and x∈V, then $\forall x\ \varphi$ is a formula.
- If φ is a formula and x∈V, then $\exists x\ \varphi$ is a formula.
- If φ is a formula and α is a program, then $[\alpha]\ \varphi$ is a formula.

Free and bounded occurrences of variables

Let us consider Q as one of the two quantifier symbols, ∀ and ∃. Now if a formula Qx: φ occurs as a subformula of another formula Ψ, we say that the scope of Qx within Ψ is the formula φ. If an occurrence of a variable y in the formula Ψ is not in the scope of any quantifier Qy with the same variable y, such an occurrence of y can be called a free occurrence of y in Ψ. If we have Qy: φ as a subformula of a formula Ψ, and y occurs free in φ, then we can say that, the occurrence of y in Ψ is bound by the quantifier Qy. Let us consider the following formula as an example:

∃x: ((∀y: ∃x: q(x, y)) ∧ p(x, y, z))

In this example, the scope of the first ∃x is ((∀y: ∃x: q(x, y)) ∧ p(x, y, z)), the scope of ∀y is ∃x: q(x, y), and the scope of the last occurrence of ∃x is q(x, y). The occurrence of x in q(x, y) is bound to the latter occurrence of ∃x. The occurrence of x in p(x, y, z) is bound to the first occurrence of ∃x, but it occurs free in the subformula ((∀y: ∃x: q(x, y)) and p(x, y, z)). The occurrence of y in p(x, y, z) is free, but it's occurrence in q(x, y) is bound to the quantifier ∀y. As we do not have any quantification for the variable z, it occurs free in p(x, y, z).

Now, we can consider a variable as a free variable of a formula φ if it has a free occurrence in the formula φ. Substitution of free variables can be done by using the notation $\varphi[t_1/x_1, ..., t_n/x_n]$ or $\varphi[t_i/x_i \mid 1 \leq i \leq n]$, which denotes the formula φ with all free occurrences of x_i replaced with term t_i, 1 ≤ i ≤ n. If we consider two notations, φ[x/s, y/t] and φ[x/s][y/t], these two notations are similar unless we have at least one occurrence of y in the variable s.

If we have a program in the formula, an assignment statement of that program can also bind free variables. Let us consider an example which has a program within it.

$$n = 0 \rightarrow [x:=n+1;\ n:=x+1;\ y:=n+3]\ y=5$$

In this example of an implication formula, the occurrence of n in the assumption part is free. It also occurs freely in the first assignment statement of the program. But, the occurrence of n in the third assignment statement is not free any more because of the presence of second assignment statement, which binds the free variable n. So, if substitution takes place for the free occurrences of n in this formula, all occurrences of n will be substituted here except the occurrence of n in the third assignment statement of the program.

We can call a formula as a closed formula or sentence if it does not have any free occurrences of variables within it. In a universal closure of a formula φ, the sentence is obtained by preceding φ with enough universal quantifiers $\forall$x to bind all the free variables of φ [5].

2.3.2 Semantics

Assigning meaning to the syntactic constructs described in the previous section involves interpreting programs and formulas over a first-order model $\mathcal{M}$. Variable range is determined by the carrier of this model. The values of variables are changed in programs by applying sequences of simple assignments, and flow of control depends on the truth values of tests performed at different times during the computation.

Definition 2. *Let us consider F as a set of function symbols, and P as a set of predicate symbols. Now, a model $\mathcal{M}$ can have following data within it:*

- *The universe or carrier of the model, $|\mathcal{M}|$, which is a non-empty set.*
- *For each function symbol in the language $f \in F$, which is of arity n, there is a n-ary function $m_{\mathcal{M}}(f)$: $\mid \mathcal{M} \mid^n \rightarrow |\mathcal{M}|$*
- *For each predicate symbol in the language $p \in P$, which is of arity n, there is a subset, $m_{\mathcal{M}}(p) \subseteq \mid \mathcal{M} \mid^n$*

In a model, a constant symbol is interpreted by one of it's elements. Moreover, a 0-ary predicate symbol can be mapped to subsets of $\mid \mathcal{M} \mid^0$, which is a set that has only one element within it. There are exactly two subsets of this set: the set itself,

and the empty set, modelling true and false, respectively.

In this system, we have considered only one model, the set of integer numbers $\mathbb{Z}$, to give semantics to the syntactic constructs of the system. In this model, integer variables are the elements of the set of variables and integer numbers are used as constant values. In addition, in this system, the meaning of each of the predicate symbols or function symbols is fixed. The usual interpretation of arithmetic operators +, -, *, /, and relational operators $<$, $>$, $\leq$, $\geq$ is used here to give meaning to a function or a predicate symbol. So, for example, the binary function symbol + is interpreted by adding two integer numbers and the value it returns is also an integer number. In this thesis, we will be using the notation $\mathcal{Z}$ to denote the standard model based on the set of integers $\mathbb{Z}$, i.e., $\mid \mathcal{Z} \mid = \mathbb{Z}$, and $m_{\mathcal{Z}}$ for giving meaning to the syntactic constructs of this system over the model of the set of integer numbers $\mathbb{Z}$.

Definition 3. *Let us consider $\mathcal{Z}$, the model of integer numbers. A valuation u: V $\rightarrow |\mathcal{Z}|$ is a function from the set of variables V to the universe of the model. For a valuation u, a variable x, and a value $a \in |\mathcal{Z}|$, the valuation denoted by u[a/x] is defined by:*

$$u[a/x](y) =^{def} \begin{cases} a & \textit{iff } x{=}y \\ u(y) & \textit{iff } x{\neq}y \end{cases}$$

where y in V.

Definition 4. *Let us consider $\mathcal{Z}$ be the model of integer numbers, and u be a valuation. The extension $\bar{u}$: Term $\rightarrow |\mathcal{Z}|$ is defined by:*

- *$\bar{u}(x) = u(x)$ for every $x \in V$.*
- *$\bar{u}(f(t_1, \ldots, t_n)) = m_{\mathcal{Z}}(\bar{u}(t_1), \ldots, \bar{u}(t_n))$,*
  where, $t_i \in$ Term, for $1 \leq i \leq n$.

States as Valuations

Values of program variables determine all relevant information at any moment during the computation. Therefore, states are represented by valuations u, v etc. of the variables V over the carrier of the model $\mathcal{Z}$. As we have used the set of integers as the model of this system, all valuations in this system map a variable to an integer number. So, all formulas of this system are interpreted over the set of integers, and

all program variables store integer numbers. We denote the set of all valuations or states over the model $\mathcal{Z}$ by $S^{\mathcal{Z}}$.

Assuming u as the beginning valuation, after the execution of the program α if we halt in a valuation v, then we can associate the pair (u,v) of such valuations with the program α. In this case, (u,v) will be referred as an input/output pair of α and will be written as (u,v) $\in m_{\mathcal{Z}}(\alpha)$. As we mentioned, in this system, a valuation maps a set of program variables to a set of integer numbers. So, a set of integer numbers is used here to represent each element of the input/output pair of a program α, for finding the value of a term, and for evaluating the truth value of a formula. Let us consider a formula as an example:

$$\text{x=0} \wedge \text{y=2} \rightarrow [\text{x=x+y; y=x*y}]\ \text{z=y-x}$$

A general input/output pair for the program of the given formula can be given as, $(u, u[u(x) + u(y)/x][u[u(x) + u(y)/x](x) * u[u(x) + u(y)/x](y)/y]) \in m_{\mathcal{Z}}(\alpha)$, where, $u[u(x) + u(y)/x][u[u(x) + u(y)/x](x) * u[u(x) + u(y)/x](y)/y] = u[u(x) + u(y)/x]$ $[(u(x) + u(y)) * u(y)/y]$.
Now, if u satisfies the left-side of the formula, i.e., if u(x) = 0 and u(y) = 2, then after executing the program we can get,
$u[u(x) + u(y)/x][(u(x) + u(y)) * u(y)/y](x) = u(x) + u(y) =$ 0+2 = 2, and
$u[u(x) + u(y)/x][(u(x) + u(y)) * u(y)/y](y) = (u(x) + u(y)) * u(y)$=(0+2)*2=4.
So, this state satisfies z=y-x if $u(z)$=2.

Assignment Statements

Every program is associated with a binary relation which is called the input/output relation,

$$m_{\mathcal{Z}}(\alpha) \subseteq S^{\mathcal{Z}} \times S^{\mathcal{Z}},$$

and every formula φ is associated with a set,

$$m_{\mathcal{Z}}(\varphi) \subseteq S^{\mathcal{Z}}.$$

These two sets are defined by mutual induction on the structure of φ and α. The semantics of the assignment statement given below will be used as the basis of the inductive definition of programs and formulas:

- Meaning of the assignment statement x:=t is given by the following binary

relation:

$$m_{\mathcal{Z}}(\text{x:=t}) =^{def} \{(\text{u,u}[\bar{u}(\text{t})/\text{x}]) \mid \text{u} \in S^{\mathcal{Z}}\}$$

Programs and Formulas

Interpretation of compound programs and formulas are given by mutual induction on the structure of program α and formula φ. As we mentioned earlier, if we consider u and v as the beginning valuation and the ending valuation of the program α respectively, we can write the pair (u,v) as,

$$(\text{u,v}) \in m_{\mathcal{Z}}(\alpha), \text{ where u} \in S^{\mathcal{Z}}, \text{ v} \in S^{\mathcal{Z}}.$$

The semantics of a program is the set of all possible input/output pairs. On the other hand, the semantics of the formula φ is comprised of the set of all valuations for which the formula is true. If the formula is true for a valuation u, we can write,

$$\text{u} \in m_{\mathcal{Z}}(\varphi), \text{ where u} \in S^{\mathcal{Z}}.$$

Let us consider the following formula φ as an example:

```
x=0 →
[ y=4;
while(¬(x>3)) do
        x=x+1;
        y=x*y;
od
]
 y=5
```

Let us consider a valuation u that satisfies the left-hand side of the implication, i.e., u(x)=0. After executing the program we obtain the valuation u[3/x][24/y], since (u,u[3/x][24/y]) is an input/output pair of the program. So, after the execution of this program the content of y is 24. As this does not satisfy the given formula, we can write,

$$\text{u} \notin m_{\mathcal{Z}}(\varphi).$$

Semantic definitions for the constructs of program and formula can be given as follows:

- $m_{\mathcal{Z}}(\alpha;\beta) = m_{\mathcal{Z}}(\alpha) \circ m_{\mathcal{Z}}(\beta) = \{(\text{u, v}) \mid \exists \text{w}(\text{u, w}) \in m_{\mathcal{Z}}(\alpha) \text{ and } (\text{w,v}) \in m_{\mathcal{Z}}(\beta)\}$.
- $m_{\mathcal{Z}}(\text{while } \varphi \text{ do } \alpha \text{ od}) = \{(\text{u, v}) \mid \exists \text{n} \geq 0\ \exists u_0, \ldots, u_n\text{: u} = u_0, \text{ v} = u_n \text{ for } u_i \in m_{\mathcal{Z}}(\varphi) \text{ and } (u_i, u_{i+1}) \in m_{\mathcal{Z}}(\alpha) \text{ for } 0 \leq \text{i} < \text{n, and } u_n \notin m_{\mathcal{Z}}(\varphi)\}$

- $m_{\mathcal{Z}}$(if φ then α else β fi) =

$$\left\{ \begin{array}{ll} (u,v), & \text{where u} \in m_{\mathcal{Z}}(\varphi) \text{ and (u, v)} \in m_{\mathcal{Z}}(\alpha) \\ (u,w), & \text{where u} \notin m_{\mathcal{Z}}(\varphi) \text{ and (u, w)} \in m_{\mathcal{Z}}(\beta) \end{array} \right\}$$

- $m_{\mathcal{Z}}(\text{false}) =^{def} \emptyset$
- $m_{\mathcal{Z}}(\text{true}) =^{def} S^{\mathcal{Z}}$
- $m_{\mathcal{Z}}(\text{a} > \text{b}) =^{def} \{\text{u} \mid \forall \text{a, b: u(a)}>\text{u(b)} \}$
- $m_{\mathcal{Z}}(\text{a} < \text{b}) =^{def} \{\text{u} \mid \forall \text{a, b: u(a)}<\text{u(b)} \}$
- $m_{\mathcal{Z}}(\text{a} \geq \text{b}) =^{def} \{\text{u} \mid \forall \text{a, b: u(a)}\geq\text{u(b)} \}$
- $m_{\mathcal{Z}}(\text{a} \leq \text{b}) =^{def} \{\text{u} \mid \forall \text{a, b: u(a)}\leq\text{u(b)} \}$
- $m_{\mathcal{Z}}(\text{a} = \text{b}) =^{def} \{\text{u} \mid \forall \text{a, b: u(a)} = \text{u(b)} \}$
- $m_{\mathcal{Z}}(\neg\varphi) =^{def} \{\text{u} \mid \text{u} \notin m_{\mathcal{Z}}(\varphi) \}$
- $m_{\mathcal{Z}}(\varphi \wedge \psi) =^{def} \{\text{u} \mid \text{u} \in m_{\mathcal{Z}}(\varphi) \text{ and u} \in m_{\mathcal{Z}}(\psi)\}$
- $m_{\mathcal{Z}}(\varphi \vee \psi) =^{def} \{\text{u} \mid \text{u} \in m_{\mathcal{Z}}(\varphi) \text{ or u} \in m_{\mathcal{Z}}(\psi)\}$
- $m_{\mathcal{Z}}(\varphi \rightarrow \psi) =^{def} \{\text{u} \mid \text{if u} \in m_{\mathcal{Z}}(\varphi)\text{, then u} \in m_{\mathcal{Z}}(\psi)\}$
- $m_{\mathcal{Z}}(\varphi \leftrightarrow \psi) =^{def} \{\text{u} \mid \text{if u} \in m_{\mathcal{Z}}(\varphi)\text{, then u} \in m_{\mathcal{Z}}(\psi)\text{, and vice versa}\}$
- $m_{\mathcal{Z}}(\forall \text{x } \varphi) =^{def} \{\text{u} \mid \forall \text{a} \in \mathbb{Z}\text{, u[a/x]} \in m_{\mathcal{Z}}(\varphi)\}$
- $m_{\mathcal{Z}}(\exists \text{x } \varphi) =^{def} \{\text{u} \mid \exists \text{a} \in \mathbb{Z}\text{, u[a/x]} \in m_{\mathcal{Z}}(\varphi)\}$
- $m_{\mathcal{Z}}([\alpha]\ \varphi) =^{def} \{\text{u} \mid \forall v \text{ if (u,v)} \in m_{\mathcal{Z}}(\alpha) \text{ then v} \in m_{\mathcal{Z}}(\varphi)\}$

2.3.3 Satisfiability and Validity of Formulas

Let $\mathcal{Z}$ be the model of integer numbers and u be a state in $S^{\mathcal{Z}}$. Now for a formula φ,

- u $\models \varphi$, if u $\in m_{\mathcal{Z}}(\varphi)$.
- $\models \varphi$, if u $\models \varphi$ for all u $\in S^{\mathcal{Z}}$.
- $\Gamma \models \varphi$, if for a valuation u $\in S^{\mathcal{Z}}$, the property u $\models \beta$ for all $\beta \in \Gamma$ implies u $\models \varphi$.

2.4 Reasoning in FODL

Reasoning in first-order dynamic logic can be performed in two ways: uninterpreted reasoning and interpreted reasoning. Reasoning in the uninterpreted level deals with the properties that can be expressed in the logic that are independent of the domain of interpretation. In contrast, reasoning in the interpreted level involves the use of the logic which depends on a particular domain or a limited class of domains. In this level, syntactically programs and formulas are the same as on the uninterpreted level except the assumption of a fixed model or class of models. Properties of particular models, over which programs are interpreted, determine the computational behavior of programs. In this system, we have used an interpreted version as this level of reasoning is very close to the actual process of reasoning about concrete, fully specified programs. The interpreted level of reasoning includes almost any task of verification of the correctness of an actual program. We have considered a specific model of integer numbers with the usual arithmetic operations.

Let us consider '-' as the first-order definable operation of subtraction and gcd(x,y) as the first-order definable operation of giving the greatest common divisor of x and y. Then, we can consider the following formula as $\mathcal{Z}$-valid DL formula:

$$\text{x}=x_1 \wedge \text{y}=y_1 \wedge \text{x*y} \geq 1 \rightarrow [\alpha]\ (\text{x}=\text{gcd}(x_1, y_1))$$

where, α is the following regular program:

```
while(¬(x=y)) do
    if(x>y) then
        x=x-y
    else
        y=y-x
od
```

This formula states the partial correctness of an actual program for computing the greatest common divisor which is interpreted over model $\mathcal{Z}$.
Let us consider another example of formula over $\mathcal{Z}$:

```
∀x: x≥1 →
 [ while(x>1) do
        if(even(x)) then
            x=x/2
        else
            x=3*x+1
```

```
            fi
        od
    ]
    x=1
```

where integer division is represented by /, and the relation even() checks whether it's argument is even or odd. This formula corresponds to a famous problem in number theory, namely, `the 3x+1 problem` [5, 24]. In this formula, if we start with an arbirary positive integer, then after executing the following two operations repeatedly, we will eventually reach 1:

- if the number is even, it will be divided by 2.
- if the number is odd, it will be multiplied by 3, and then 1 will be added to the result of multiplication.

We can generalize the specific model $\mathcal{Z}$, which will result in a class of arithmetical models. we can define a model $\mathcal{Z}$ as arithmetical if following are true:

- It has a copy of $\mathcal{Z}$ that is first-order definable.
- It has first-order definable functions that will code the finite sequences of elements of the model $\mathcal{Z}$ into single elements and also has first-order definable functions for corresponding decoding.

Importance of arithmetic models is significant because of the following facts:

- Most models emerging naturally in computer science are arithmetical. For example, discrete models with recursively defined data types are arithmetical.
- Addition of appropriate encoding and decoding functionalities can extend any model to an arithmetical one.

Chapter 3

Calculus of the Deductive System

In this chapter we introduce the calculus of the deductive system for first-order dynamic logic. In the next section we explore the proof rules we have implemented in this deductive system. Then we will prove the soundness of these rules.

3.1 Inference Rules

In this system, the main design criteria is the educational use of the system. For this reason, we designed the rules of this deductive system in a way so that the verification procedure in this system is very close to the procedure of program property verification using Hoare logic by hand. For example, as we start the verification from the bottom of the program and move backwards, for the assignment rule, we have the derivation, $\Gamma \vdash [p]\ \Psi[a\ /\ x]$, as our assumption, and our assertion/goal is the derivation, $\Gamma \vdash [p;\ x{:=}a]\ \Psi$.

A popular approach to transform programs and specifications to first-order verification conditions is the weakest preconditions predicate transformer [2]. Let us consider a program `p` and a formula `post` representing a postcondition formula. The weakest preconditions transformer constructs a formula `wp(p, post)`, which is the weakest precondition of `p` with respect to `post`. If `wp(p,post)` holds before `p`, then `post` holds afterwards. Any other formulas having this property are stronger than the weakest precondition `wp(p, post)`, that means, they logically imply `wp(p, post)`. Many of the popular program verifiers namely, ES-C/Java2, Boogie, JACK and Why use the weakest preconditions for generating the verification conditions [10, 25, 26].

Let us consider an assignment statement `a:=t`. For this assignment statement, we can

constuct the weakest precondition with respect to a postcondition post as wp(a:=t, post)=post[t/a], where post[t/a] denotes the the result of substituting all instances of a by t in post. For example, wp(a:=a+1, a:=3)=(a+1:=3) states that, if the value of the variable a after the assignment is to be 3, then the term a+1 must have to be equal to 3 before the assignment. The weakest precondition of the sequential composition p1;p2 of programs p1 and p2 can be computed as wp(p1;p2, post) = wp(p1, wp(p2, post)). It means, at first wp transformer is applied to p2 and post, and then the outcome of this application is used as the postcondition which is to be established by p1.

We can use the weakest precondition approach in dynamic logic in a natural way. The dynamic logic formula [p] post is the weakest precondition of p with respect to post, not in a first-order form though. This dynamic logic formula can be rewritten to a first-order form by starting from the back and applying assignments to the postcondition as substitutions in the above mentioned way. For example, if we consider a formula [a:=t; b:=$t^{'}$] post, we can first transform it into [a:=t] post[$t^{'}$/b] and then into (post[$t^{'}$/b])[t/a].

An alternative to the weakest preconditions is the strongest postconditions predicate transformers. If we have a precondition formula pre and a program p, this transformer constructs a formula sp(pre, p) which is guaranteed to be hold after the execution of p if we start the execution in a state satisfying pre. This strongest postcondition implies all other formulas with this property. In the KIV tool, the implementation of dynamic logic calculus for java programs is based on a variation of strongest postconditions.

We can derive the strongest postcondition of an assignment a:=t as sp(pre, a:=t)=$\exists a^{'}$; (pre[$a^{'}$/a] $\cap$ a: = t[$a^{'}$/a]). It means there must be some previous value $a^{'}$ of a such that if we use $a^{'}$ for a then pre holds and the new value of a will be the previous value of t. In other words, sp(pre, a:=t)=(pre[$a^{'}$/a] $\cap$ a := t[$a^{'}$/a]) can be defined, where $a^{'}$ is a fresh function symbol. For example, we can consider sp(a:=2, a:=a+1) = ($a^{'}$ = 2 $\cap$ a: = $a^{'}$ +1). Managing sequential composition p1;p2 involves first the application of sp to p1 and then to p2. That means, sp(pre, p1;p2) = sp(sp(pre, p1), p2).

The advantage of using the weakest precondition transformation over the strongest

postcondition transformation is that it produces smaller verification conditions, which encourages us to follow the weakest precondition approach. For sp, we need to introduce an existentially quantified symbol a' for denoting the pre-assignment value of the changed variable, whereas for wp, an assignment amounts to a mere substitution in the postcondition.

Here, the first-order calculus is formulated in a sequent style. We will have exactly one formula in the right-hand side. A logical calculus usually makes use of proof rules to infer a conclusion from a finite set of assumptions. Given a sequence of formulas Γ_1 and a derivation $\Gamma_1 \vdash \Psi_1$, application of a proof rule of the calculus to the derivation will result in a new derivation $\Gamma_2 \vdash \Psi_2$. In the following step, applying a proof rule in the resultant derivation will generate a new derivation $\Gamma_3 \vdash \Psi_3$. If we apply the proof rule continuously, finally we will get the conclusion $\Gamma \vdash \Psi$. The resulting derivation is actually a tree where assumptions are represnted as leaves, applications of rules are represented as intermediate nodes, and the conclusion is represented as the root of the tree. The inference rules of our deductive system are listed below. The sequencing and consequence rules of Hoare logic are implicitly incorporated in the loop rule of our calculus.

In this system, equational reasoning is handled in a special way in order to make the reasoning close to the actual human reasoning. Users of this system have the privilege to choose a particular axiomatic rule from a list of applicable rules to initiate the process of equational reasoning. This feature is implemented in a way so that one will experience the flavor of doing the equational reasoning with a pen and a piece of paper. An example of equational reasoning is demonstrated in Chapter 4.

As we mentioned earlier, the implemented inference rules for reasoning about programs are equivalent to Hoare logic rules, and the structure of these rules is very close to the actual Hoare logic rules, which makes it very simple to understand the techniques of program property verification. The combination of these features of DL rules, FOL rules, and equational reasoning make this system very well suited for educational purposes. As an interactive program verification tool, user interactions are required for the application a proof rule to proceed to the next step of verification. The key feature of this system is the visual explanation of the application of these rules, which makes it a convenient one to the learner of the program property verification techniques.

Inference rules are applied for carrying out the logical operations. There are three different types of rules we have in our calculus:

- Model specific rules and axioms
- Rules for first-order reasoning
- Rules for dynamic reasoning

Model specific rules and axioms include the induction rule, and the axiomatic rules for the set of integer numbers, $\mathbb{Z}$. Mathematical induction is a method of mathematical proof which is implemented to establish that a given statement is true for all integer numbers.

$$\frac{\Gamma \vdash \varphi[0/x] \ \Gamma,\varphi \vdash \varphi[x+1/x] \ \Gamma,\varphi \vdash \varphi[x-1/x]}{\Gamma \vdash \varphi} \quad \text{Induction}$$

Table 3.1: Induction Rule

As we mentioned earlier, a derivation is actually a tree with the premises as leaves, applications of rules as nodes, and the conclusion as the root. This tree will be ended at axiom rules which do not have any premises. The axiomatic rules that have been listed here are the axioms we currently have in this system. This list can be extended with the newer axioms, if equational reasoning of a term or formula requires the application of those axioms.

Table 3.2: Axiomatic rules for integer numbers

$$\frac{}{\forall a\colon a = a+0}$$

$$\frac{}{\forall a\colon a*1 = a}$$

$$\frac{}{\forall a\colon a*0 = 0}$$

$$\frac{}{\forall a\colon \forall b\colon a+b = b+a}$$

$$\frac{}{\forall a\colon \forall b\colon a*b = b*a}$$

$$\frac{}{\forall a\colon \forall b\colon \forall c\colon a+(b+c) = (a+b)+c}$$

$$\frac{}{\forall a\colon \forall b\colon \forall c\colon a*(b*c) = (a*b)*c}$$

$$\frac{}{\forall a\colon \forall b\colon \forall c\colon a*(b+c) = (a*b)+(a*c)}$$

$$\frac{}{\forall a\colon \forall b\colon \forall c\colon (a+b)*c = (a*c)+(b*c)}$$

Continued on next page

Table 3.2 – continued from previous page

$$\overline{\forall a\colon \forall b\colon \forall c\colon a{*}(b{-}c) = (a{*}b){-}(a{*}c)}$$

$$\overline{\forall a\colon \forall b\colon \forall c\colon (a{-}b){*}c = (a{*}c){-}(b{*}c)}$$

$$\overline{\forall a\colon \forall b\colon a{+}b = b{+}c \leftrightarrow a = c}$$

$$\overline{\forall a\colon \forall b\colon a{+}b < b{+}c \leftrightarrow a < c}$$

$$\overline{\forall a\colon \forall b\colon a{+}b > b{+}c \leftrightarrow a > c}$$

$$\overline{\forall a\colon \forall b\colon a <= b \leftrightarrow b >= a}$$

$$\overline{\forall a\colon \forall b\colon a >= b{+}1 \rightarrow a >= b}$$

$$\overline{\forall a\colon \forall b\colon a >= b \wedge a <= b \leftrightarrow a = b}$$

$$\overline{\forall a\colon \forall b\colon a >= b \leftrightarrow \neg(a < b)}$$

$$\overline{\forall a\colon \forall b\colon a <= b \leftrightarrow \neg(a > b)}$$

Lemma 3.1.1. *If the valuations u and v coincide on all variables of t, where $t \in$ Term, then $\overline{u}(t) = \overline{v}(t)$.*

Proof: If t = x, where x ∈ V, we get,

$$\overline{u}(\mathrm{t}) = \mathrm{u}(\mathrm{x}) = \mathrm{v}(\mathrm{x}) = \overline{v}(\mathrm{t}).$$

If t = f(t_1, ..., t_n), where f is an n-ary function symbol, and t_1, ..., t_n are terms, we can conclude,

$$\begin{aligned}\overline{u}(t) =& \mathrm{m}_{\mathcal{Z}}(\overline{u}(t_1), \ldots, \overline{u}(t_n))\\ &= m_{\mathcal{Z}}(\overline{v}(t_1), \ldots, \overline{v}(t_n))\\ &= \overline{v}(\mathrm{t}).\end{aligned}$$

□

Lemma 3.1.2. *If the valuations u and v coincide on all free variables of a formula φ, then $u \models \varphi$ iff $v \models \varphi$.*

Proof: If φ = p(t_1, ..., t_n), where p is an atomic formula and t_1, ..., t_n are terms, then every variable in each of terms occurs freely in the formula φ. So, we can conclude $\overline{u}(t_i) = \overline{v}(t_i)$ for i = {1, ..., n} using Lemma 3.1.1, which implies the assertion immediately.

Left logical rules	Right logical rules
$\frac{\Gamma\vdash t_1=t_2 \;\; \Gamma\vdash\Psi[t_1/r]}{\Gamma\vdash\Psi[t_2/r]}{=}\mathrm{L}$	$\frac{}{\vdash t=t}{=}\mathrm{R}$
$\frac{\Gamma,\varphi_1,\varphi_2\vdash\Psi}{\Gamma,\varphi_1\wedge\varphi_2\vdash\Psi}\wedge\mathrm{L}$	$\frac{\Gamma\vdash\varphi_1 \;\; \Gamma\vdash\varphi_2]}{\Gamma\vdash\varphi_1\wedge\varphi_2}\wedge\mathrm{R}$
$\frac{\Gamma,\varphi_1\vdash\Psi \;\; \Gamma,\varphi_2\vdash\Psi}{\Gamma,\varphi_1\vee\varphi_2\vdash\Psi}\vee\mathrm{L}$	$\frac{\Gamma\vdash\varphi_1}{\Gamma\vdash\varphi_1\vee\varphi_2}\vee R \quad \frac{\Gamma\vdash\varphi_2}{\Gamma\vdash\varphi_1\vee\varphi_2}\vee R$
$\frac{\Gamma\vdash\varphi_1 \;\; \Gamma,\varphi_2\vdash\Psi}{\Gamma,\varphi_1\to\varphi_2\vdash\Psi}\to\mathrm{L}$	$\frac{\Gamma,\varphi_1\vdash\varphi_2}{\Gamma\vdash\varphi_1\to\varphi_1}\to\mathrm{R}$
$\frac{\Gamma,\varphi_1\to\varphi_2,\varphi_2\to\varphi_1\vdash\Psi}{\Gamma,\varphi_1\leftrightarrow\varphi_2\vdash\Psi}\leftrightarrow\mathrm{L}$	$\frac{\Gamma\vdash\varphi_1\to\varphi_2 \;\; \Gamma\vdash\varphi_2\to\varphi_1}{\Gamma\vdash\varphi_1\leftrightarrow\varphi_2}\leftrightarrow\mathrm{R}$
$\frac{\Gamma\vdash\varphi}{\Gamma,\neg\varphi\vdash\Psi}\;\neg L$	$\frac{\Gamma,\varphi\vdash\bot}{\Gamma\vdash\neg\varphi}\neg\mathrm{R}$
$\frac{\Gamma,\varphi[s/a]\vdash\Psi}{\Gamma,(\forall a)\varphi\vdash\Psi}\forall\mathrm{L}$	$\frac{\Gamma\vdash\varphi}{\Gamma\vdash(\forall a)\varphi}\forall\mathrm{R}$ If no free occurrence of a in any formulas of Γ
$\frac{\Gamma,\varphi\vdash\Psi}{\Gamma,(\exists a)\varphi\vdash\Psi}\exists\mathrm{L}$ If no free occurrence of a in any formulas of Γ and in Ψ	$\frac{\Gamma\vdash\varphi[s/a]}{\Gamma\vdash(\exists a)\varphi}\exists\mathrm{R}$
$\frac{\Gamma,\neg\varphi\vdash\bot}{\Gamma\vdash\varphi}$ PBC	
$\frac{}{\Gamma\vdash\varphi}$, if $\varphi\in\Gamma$	

Table 3.3: Logical rules for first-order reasoning

If the case φ is one of the following formulas, true, false, $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, or $\varphi_1 \to \varphi_2$, then the proof is the straight forward applications of the induction hypothesis.

Now, let us assume φ = Qx: φ', where Q $\in \{\forall, \exists\}$. The free variables of φ' includes the free variables of φ, and the variable x. Consequently, the valuations u[a/x]

Rule	Name
$\dfrac{\Gamma \vdash [p]\Psi}{\Gamma \vdash [p;\, skip]\Psi}$	skip
$\dfrac{\Gamma \vdash [p]\Psi[a/x]}{\Gamma \vdash [p;\, x{:=}a]\Psi}$	assignment
$\dfrac{\Gamma \vdash [p]((b{\rightarrow}[p1]\Psi){\wedge}(\neg b{\rightarrow}[p2]\Psi))}{\Gamma \vdash [p;\, if\ b\ then\ p1\ else\ p2\ fi]\Psi}$	if-else
$\dfrac{\Gamma \vdash [p]\varphi \quad \Gamma,\varphi{\wedge}b \vdash [p1]\varphi \quad \Gamma \vdash \varphi{\wedge}\neg b \rightarrow \Psi}{\Gamma \vdash [\ p;\, while\ b\ do\ p1\ od]\Psi}$	loop

Table 3.4: Inference rules for dynamic reasoning

and v[a/x] for an arbitrary a $\in \mid \mathcal{Z} \mid$ coincide on all free variables in $\varphi^{'}$. So, we can conclude,

$$\begin{aligned} \mathrm{u} \models \varphi &\Leftrightarrow \mathrm{u[a/x]} \models \varphi^{'}, \text{ for all/some a} \in \mid \mathcal{Z} \mid \\ &\Leftrightarrow \mathrm{v[a/x]} \models \varphi^{'}, \text{ for all/some a} \in \mid \mathcal{Z} \mid \\ &\Leftrightarrow \mathrm{v} \models \varphi. \end{aligned}$$

Here, the second equivalence is an application of the induction hypothesis.

Finally, let us assume, $\varphi = [\alpha]\ \varphi^{'}$. Now, free variables of φ includes free occurrences of variables in program α, and any free variables of formula $\varphi^{'}$, whose occurrence is not bound by any assignment statement of program α. So that, from induction hypothesis, we can conclude,

$$\mathrm{u} \models \varphi \Leftrightarrow \mathrm{v} \models \varphi.$$ □

Lemma 3.1.3. *If we consider u as a valuation, t, $t^{'} \in$ Term as terms, and φ as a formula, then,*

$$\overline{u}(t^{'}[t/x]) = \overline{u[\overline{u}(t)/x]}(t^{'}).$$

Proof: This assertion is shown by induction below:

If $t^{'}$ = y, where y $\in$ V, we distinguish two cases:

- If x = y, we can get,
  $$\overline{u}(t^{'}[\mathrm{t/x}]) = \overline{u}(\mathrm{y}) = \overline{u[\overline{u}(t)/x]}(\mathrm{x}) = \overline{u[\overline{u}(t)/x]}(t^{'}).$$
- If x $\neq$ y, the valuations u and $\overline{u[\overline{u}(t)/x]}$ coincide on all variables in $t^{'}$. Now, using Lemma 3.1.1, we can conclude,
  $$\overline{u}(\mathrm{t}^{'}[\mathrm{t/x}]) = \overline{u}y = \overline{u[\overline{u}(t)/x]}(y) = \overline{u[\overline{u}(t)/x]}(\mathrm{t}^{'}).$$

If $t' = \mathrm{f}(t_1, \ldots, t_n)$, we can get immediately,

$$\begin{aligned}\overline{u}(\mathrm{t}'[\mathrm{t/x}]) &= m_{\mathcal{Z}}(\overline{u}(t_1[\mathrm{t/x}]), \ldots, \overline{u}(t_n[\mathrm{t/x}])), \text{ by substitution}\\ &= m_{\mathcal{Z}}(\overline{u[\overline{u}(t)/x]}(t_1), \ldots, \overline{u[\overline{u}(t)/x]}(t_n)), \text{ by induction hypothesis}\\ &= \overline{u[\overline{u}(t)/x]}(t').\end{aligned}$$

□

Lemma 3.1.4. *If u is a valuation, t is a term, and φ is a formula, then $u \in m_{\mathcal{Z}}(\varphi[t/x])$ iff $u[\overline{u}(t)/x] \in m_{\mathcal{Z}}(\varphi)$.*

Proof: If $\varphi = \mathrm{p}(t_1, \ldots, t_n)$, we can conclude,

$$\begin{aligned}&\mathrm{u} \models \varphi[\mathrm{t/x}]\\ \Leftrightarrow\ &\mathrm{u} \models \mathrm{p}[t_1[\mathrm{t/x}], \ldots, t_n[\mathrm{t/x}]], \text{ by substitution}\\ \Leftrightarrow\ &(\overline{u}(t_1[\mathrm{t/x}]), \ldots, \overline{u}(t_n[\mathrm{t/x}])) \in m_{\mathcal{Z}}(p)\\ \Leftrightarrow\ &(\overline{u[\overline{u}(t)/x]}(t_1), \ldots, \overline{u[\overline{u}(t)/x]}(t_n)) \in m_{\mathcal{Z}}(p)\\ \Leftrightarrow\ &\mathrm{u}[\overline{u}(\mathrm{t})/\mathrm{x}] \models \mathrm{p}(t_1, \ldots, t_n)\\ \Leftrightarrow\ &\mathrm{u}[\overline{u}(\mathrm{t})/\mathrm{x}] \models \varphi\end{aligned}$$

If the case φ is one of the following formulas, true, false, $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, then it is straight forward applications of the induction hypothesis.
Now, let us assume $\varphi = \mathrm{Qy}\colon \varphi'$ with $\mathrm{Q} \in \{\forall, \exists\}$. Now, we will distinguish two different cases:

- If x = y, where x $\in$ V and x does not occur free in φ, then, $\varphi[\mathrm{t/x}] = \varphi$. By Lemma 3.1.2, we get, $\mathrm{u} \models \varphi$ iff, $\mathrm{u}[\overline{u}(\mathrm{t})/\mathrm{x}] \models \varphi$, and, hence, the assertion.
- If x $\neq$ y, we can conclude,

$$\begin{aligned}\mathrm{u} \models \varphi[\mathrm{t/x}] &\Leftrightarrow \mathrm{u} \models \mathrm{Qy}\colon \varphi'[\mathrm{t/x}]\\ &\Leftrightarrow \mathrm{u[a/y]} \models \varphi'[\mathrm{t/x}], \text{ for all/some a} \in |\ \mathcal{Z}\ |\\ &\Leftrightarrow \mathrm{u[a/y]}[\overline{u[a/y]}(\mathrm{t})/\mathrm{x}] \models \varphi', \text{ for all/some a} \in |\ \mathcal{Z}\ |\\ &\Leftrightarrow \mathrm{u[a/y]}[\overline{u}(\mathrm{t})/\mathrm{x}] \models \varphi', \text{ for all/some a} \in |\ \mathcal{Z}\ |, \text{ by Lemma 3.1.2,}\\ &\qquad\qquad \text{since t is free for x in Qy: } \varphi\\ &\Leftrightarrow \mathrm{u}[\overline{u}(\mathrm{t})/\mathrm{x}]\ [\mathrm{a/y}] \models \varphi', \text{ for all/some a} \in |\ \mathcal{Z}\ |, \text{ since x} \neq \text{y}\\ &\Leftrightarrow \mathrm{u}[\overline{u}(\mathrm{t})/\mathrm{x}] \models \mathrm{Qy}\colon \varphi'\\ &\Leftrightarrow \mathrm{u}[\overline{u}(\mathrm{t})/\mathrm{x}] \models \varphi.\end{aligned}$$

  Here, the third equivalence is an application of the induction hypothesis.

Finally, let us assume, $\varphi = [\alpha]\ \varphi'$. Now, free variables of φ includes free occurrences of variables in program α, and any free variables of formula φ', whose occurrence

is not bound by any assignment statement of program α. So that, from induction hypothesis, we can conclude,

$$u \in m_{\mathcal{Z}}(\varphi[t/x]) \Leftrightarrow u[\overline{u}(t)/x] \in m_{\mathcal{Z}}(\varphi). \qquad \square$$

3.2 Soundness of the inference rules

Soundness of a deductive system enforces that whatever can be derived in the deductive system is also valid. That is,

$$\Gamma \vdash \varphi \rightarrow \Gamma \models \varphi.$$

where, Γ is a finite list of first-order formulas φ_1, ..., φ_n, and φ is a first-order formula. The order and multiple occurrences of a formula within Γ are treated implicitly. By the notation, Γ, φ, we donote the list Γ extended by a first-order formula φ.

This is the least requirement of any logical system. The following lemma will illustrate the soundness of our first-order calculus.

Theorem 3.2.1. *Let Γ be a sequence of first-order formulas φ_1, ..., φ_n and Ψ be a first-order formula. If $\Gamma \vdash \Psi$ is valid, then $\Gamma \models \Psi$ holds.*

Proof: The proof is done by induction on the derivation $\Gamma \vdash \Psi$.

Base cases: We have several base cases. All the axioms we have for the integer numbers are true because of the choice of the model, i.e., the set of integers. The axioms of the table above are trivial. $\square$

$\forall$**I:** Here, $\Psi = \forall$x: Ψ', and the derivation we have is φ_1, ..., $\varphi_n \vdash \Psi'$. The variable condition implies that there is no free occurrence of x in any of φ_1, ..., φ_n. Let us consider $\mathcal{Z}$ be the model of integer numbers and u be a valuation with u $\models \varphi_i$ for i $\in$ { 1, ..., n}. Using the Lemma 3.1.2, we can have, u[a/x] $\models \varphi_i$ for all a $\in$ | $\mathcal{Z}$ | and i $\in$ {1, ..., n}. Now, using the induction hypothesis we can conclude, u[a/x] $\models \Psi'$ for all a $\in$ | $\mathcal{Z}$ |, and therefore, u $\models \Psi$. $\square$

$\forall$**E:** In this case, $\Psi = \Psi'$[t/x], and we have a derivation φ_1, ..., $\varphi_n \vdash \forall$x: Ψ'. Now let us assume u be a valuation with u $\models \varphi_i$ for i $\in$ {1, ..., n}. So, by induction hypothesis we can conclude, u $\models \forall$x: Ψ', and, in particular, u[$\overline{u}$(t)/x] $\models \Psi'$. So we can get, u $\models$ Ψ'[t/x]. $\square$

∃I: Let us consider the case of a finite subset of the set of integer numbers, i.e., a model $\{a_1, ..., a_n\}$. In this case, an existential formula ∃x: φ is true if it is true for one of the integer elements a_1, ..., a_n. Assuming the terms t_1, ..., t_n denote the elements, i.e., $\overline{u}(t_i)=a_i$, ∃x: φ is true if $\varphi[t_1/x] \vee ... \vee \varphi[t_n/x]$ is true. As a result, we get a rule similiar to ∨ elimination. The formula χ must be independent of x, and x be local to that subtree motivating the variable condition of this rule.

□

∃E: Let us consider the case of a finite subset of integer numbers, i.e., a model $\{a_1, ..., a_n\}$. In this case, an existential formula ∃x: φ is true if it is true for one of the integer elements a_1, ..., a_n. Assuming the terms t_1, ..., t_n denote the elements, i.e., $\overline{u}(t_i)=a_i$, ∃x: φ is true if $\varphi[t_1/x] \vee ... \vee \varphi[t_n/x]$ is true. As a result, we get a rule similiar to ∨ elimination. The formula χ must be independent of x, and x be local to that subtree motivating the variable condition of this rule.

$$\frac{\exists x{:}\varphi \quad \begin{matrix}[\varphi] \\ \vdots \\ \chi\end{matrix}}{\chi}\ \exists E.$$

□

=L: By the induction hypothesis, we have $\Gamma \models t_1 = t_2$ and $\Gamma \models \Psi[t_1/x]$. Let u be a valuation so that u $\in m_{\mathcal{Z}}(\beta)$ for all $\beta \in \Gamma$. Then $\overline{u}(t_1) = \overline{u}(t_2)$ and u $\in m_{\mathcal{Z}}(\Psi[t_1/x])$. From the Lemma 3.1.4, we obtain u$[\overline{u}(t_1)/x] \in m_{\mathcal{Z}}(\Psi)$. This implies u$[\overline{u}(t_2)/x] \in m_{\mathcal{Z}}(\Psi)$, and, hence u $\in m_{\mathcal{Z}}(\Psi[t_2/x])$ using the Lemma 3.1.4 again.

□

∧L: In this case, $\Gamma = \Gamma', \varphi_1 \wedge \varphi_2$ and we have a derivation $\Gamma', \varphi_1, \varphi_2 \vdash \Psi$. Now, assuming u be a valuation, u $\models \Gamma'$ and u $\models \varphi_1 \wedge \varphi_2$ holds. From the definition of $\models$ we get, u $\models \varphi_1 \wedge \varphi_2$ iff u $\models \varphi_1$ and u $\models \varphi_2$. Which implies u $\models \Gamma'$, u $\models \varphi_1$ and u $\models \varphi_2$ holds. By the induction hypothesis we can conclude u $\models \Psi$, and hence, $\Gamma', \varphi_1 \wedge \varphi_2 \models \Psi$. □

∨L: In this case, $\Gamma = \Gamma', \varphi_1 \vee \varphi_2$ and we have derivations $\Gamma', \varphi_1 \vdash \Psi$ and $\Gamma', \varphi_2 \vdash \Psi$. Now, assuming u be a valuation, u $\models \Gamma'$ and u $\models \varphi_1 \vee \varphi_2$ holds. From the definition of $\models$ we get, u $\models \varphi_1$ or u $\models \varphi_2$. In the first case we conclude, $\models$ satisfies Γ', φ_1. Using induction hypothesis we conclude, u $\models \Psi$. □

$\rightarrow$**L:** In this case, $\Gamma = \Gamma', \varphi_1 \rightarrow \varphi_2$ and we have derivations $\Gamma' \vdash \varphi_1$ and $\Gamma', \varphi_2 \vdash \Psi$. Now, assuming u be a valuation, u $\models \Gamma'$ and u $\models \varphi_1 \rightarrow \varphi_2$ holds. From the definition of $\models$ we get, u $\models \varphi_1 \rightarrow \varphi_2$ iff u $\models \varphi_2$ or u $\not\models \varphi_1$. □

$\leftrightarrow$**L:** In this case, $\Gamma = \Gamma', \varphi_1 \leftrightarrow \varphi_2$ and we have derivations $\Gamma', \varphi_1 \rightarrow \varphi_2, \varphi_2 \rightarrow \varphi_1 \vdash \Psi$. Now assuming u be a valuation, u $\models \Gamma'$ and u $\models \varphi_1 \leftrightarrow \varphi_2$ holds. From the definition of $\models$ we get, u $\models \varphi_1 \leftrightarrow \varphi_2$ iff u $\models \varphi_1 \rightarrow \varphi_2$ and u $\models \varphi_2 \rightarrow \varphi_1$. □

$\neg$**L:** In this case, $\Gamma = \Gamma', \neg\varphi$ and we have a derivation $\Gamma' \vdash \varphi$. Now, assuming u be a valuation, u $\models \Gamma'$ and u $\models \neg\varphi$ holds and by induction hypothesis we get, u $\models \varphi$. As this is a contradiction, no such u exists. So, $\Gamma', \neg\varphi \models \Psi$ is always true. □

=**R:** It is trivial. □

$\wedge$**R:** In this case $\Psi = \varphi_1 \wedge \varphi_2$ and we have derivations $\Gamma \vdash \varphi_1$ and $\Gamma \vdash \varphi_2$. Now, assuming u be a valuation, u $\models \Gamma$ holds. From the induction hypothesis we can conclude, u $\models \varphi_1$ and u $\models \varphi_2$, which implies, u $\models \varphi_1 \wedge \varphi_2$, and hence u $\models \Psi$. □

$\vee$**R:** In this case $\Psi = \varphi_1 \vee \varphi_2$ and we have a derivation $\Gamma \vdash \varphi_1$. Now, assuming u be a valuation, u $\models \Gamma$ holds. By the induction hypothesis we can conclude, u $\models \varphi_1$, and by the definition of $\models$ we can get u $\models \varphi_1 \vee \varphi_2$. □

$\rightarrow$**R:** In this case $\Psi = \varphi_1 \rightarrow \varphi_2$ and we have a derivation $\Gamma, \varphi_1 \vdash \varphi_2$. Now, assuming u be a valuation, u $\models \Gamma$ holds. If u $\models \varphi_1$ holds then from the induction hypothesis we can get, u $\models \varphi_2$. By the definition of $\models$ we can conclude u $\models \varphi_1 \rightarrow \varphi_2$. If u $\not\models \varphi_1$ then by the definition of $\models$ we can get u $\models \varphi_1 \rightarrow \varphi_2$. □

$\leftrightarrow$**R:** In this case $\Psi = \varphi_1 \leftrightarrow \varphi_2$ and we have two derivations, $\Gamma \vdash \varphi_1 \rightarrow \varphi_2$, and $\Gamma \vdash \varphi_2 \rightarrow \varphi_1$. Now, assuming u be a valuation, u $\models \Gamma$ holds. Then from the induction hypothesis we can get, u $\models \varphi_1 \rightarrow \varphi_2$, and u $\models \varphi_2 \rightarrow \varphi_1$. So, by the definition of $\models$ we can conclude, u $\models \varphi_1 \leftrightarrow \varphi_2$. □

$\neg$**R:** In this case $\Psi = \neg\varphi$ and we have a derivation $\Gamma, \varphi \vdash \bot$. Now, assuming $\mathcal{Z}$ be a model of integer numbers and u be a valuation, u $\models \Gamma$ holds. As $\mathcal{Z}$ satisfies φ we can conclude u $\models \bot$ from the induction hypothesis. But, this is a contradiction. So, we can conclude u $\models \neg\varphi$. □

PBC: In this case we have a derivation $\Gamma, \neg\Psi \vdash \bot$. Now, assuming u be a valuation, u $\models \Gamma$ holds. If u $\models \neg\Psi$, we can conclude u $\models \bot$ from the induction hypothesis. But, this is a contradiction. So, we conclude u $\models \Psi$. □

Induction: To prove the soundness of the induction rule we will have to show that,

If $\Gamma \models \varphi[0/x]$,
$\Gamma, \varphi \models \varphi[x+1/x]$, and
$\Gamma, \varphi \models \varphi[x-1/x]$, then the following holds:
$\Gamma \models \varphi$.

Proof: Let us assume u $\in m_{\mathcal{Z}}(\beta)$ for all $\beta \in \Gamma$. We have to show that u $\in m_{\mathcal{Z}}(\varphi)$. We distinguish three cases. If u(x) = 0, then we have u $\in m_{\mathcal{Z}}(\varphi)$ since u $\in m_{\mathcal{Z}}(\varphi$[0/x]) by the first assumption. Now let us assume u(x) > 0, and define $u^{'}$ by $u^{'}$(y) = u(y) for y $\neq$ x and $u^{'}$(x) = u(x)-1. Then u $\in m_{\mathcal{Z}}(\varphi)$ if $u^{'} \in m_{\mathcal{Z}}(\varphi$[x+1/x]). By the second assumption the latter is implied by $u^{'} \in m_{\mathcal{Z}}(\varphi)$ and $u^{'} \in m_{\mathcal{Z}}(\beta)$ for all $\beta \in \Gamma$. We iterate the previous argument u(x) times so that we finally can apply the first assumption. The case u(x) < 0 is shown analogously. □

An Inference rule for reasoning about program is sound when validity of it's premises implies the validity of it's conclusion.

Skip: To prove the soundness of the skip rule we will have to show that,

If $\Gamma \models$ [p] Ψ, then the following holds:
$\Gamma \models$ [p; skip] Ψ.

Proof: Let us assume u $\in m_{\mathcal{Z}}(\varphi)$ for all $\varphi \in \Gamma$. We have to show that u $\in m_{\mathcal{Z}}$([p;skip] Ψ). By the induction we have $\Gamma \models$ [p] Ψ, i.e., we have u $\in m_{\mathcal{Z}}$([p] Ψ). By the definition of $m_{\mathcal{Z}}$ this implies that we have for all v that if (u, v) $\in m_{\mathcal{Z}}$(p), then v $\in m_{\mathcal{Z}}(\Psi)$. Since $m_{\mathcal{Z}}$(skip) = {(u,u) | u is a valuation}, we can get $m_{\mathcal{Z}}$(p) = $m_{\mathcal{Z}}$(p; skip) so that we have, if (u, v) $\in m_{\mathcal{Z}}$(p; skip), then v $\in m_{\mathcal{Z}}(\Psi)$ for all v, i.e. u $\in m_{\mathcal{Z}}$([p; skip] Ψ). □

Assignment: To prove the soundness of the assignment rule we will have to show that,

If $\Gamma \models$ [p] Ψ[a/x], then the following holds:

$\Gamma \models$ [p; x := a] Ψ

Proof: Let us assume u $\in m_{\mathcal{Z}}(\varphi)$ for all $\varphi \in \Gamma$. We have to show that u $\in m_{\mathcal{Z}}$([p; x:=a] Ψ). By induction, we have, $\Gamma \models$ [p] Ψ[a/x], i.e., u $\in m_{\mathcal{Z}}$([p] Ψ[a/x]).
By definition of $m_{\mathcal{Z}}$ this implies that, for all v, if (u,v) $\in m_{\mathcal{Z}}$(p), then v $\in m_{\mathcal{Z}}$(Ψ[a/x]). This is equivalent to v[a/x] $\in m_{\mathcal{Z}}(\Psi)$. Now, since $m_{\mathcal{Z}}$(x:=a)={(v, v[x/a]) | v is a valuation}, we get v $\in m_{\mathcal{Z}}$([x:=a] Ψ), and, hence, u $\in m_{\mathcal{Z}}$([p; x:=a] Ψ). □

If-else: To prove the soundness of the if-else rule we will have to show that,

If $\Gamma \models$ [p] ((b $\rightarrow$ [p1] Ψ) $\wedge$($\neg$(b) $\rightarrow$ [p2] Ψ)), then the following holds:

$\Gamma \models$ [p; if b then p1 else p2 fi] Ψ.

Proof: Let us assume u $\in m_{\mathcal{Z}}(\varphi)$ for all $\varphi \in \Gamma$. We have to show that u $\in m_{\mathcal{Z}}$([p; if b then p1 else p2 fi] Ψ). By induction, we have, $\Gamma \models$ [p] ((b $\rightarrow$ [p1] Ψ) $\wedge$ ($\neg$b $\rightarrow$ [p2] Ψ)). So, u $\in m_{\mathcal{Z}}$([p] ((b $\rightarrow$ [p1] Ψ) $\wedge$ ($\neg$b $\rightarrow$ [p2] Ψ))). By definition of $m_{\mathcal{Z}}$, we have for all y that if (u,y) $\in m_{\mathcal{Z}}$(p), then y $\in m_{\mathcal{Z}}$((b $\rightarrow$ [p1] Ψ) $\wedge$ ($\neg$b $\rightarrow$ [p2] Ψ)). The latter is equivalent to,

1. y $\in m_{\mathcal{Z}}$(b $\rightarrow$ [p1] Ψ).

2. y $\in m_{\mathcal{Z}}$($\neg$b $\rightarrow$ [p2] Ψ).

Now, let us distinguish two cases. If y $\in m_{\mathcal{Z}}$(b), then y $\in m_{\mathcal{Z}}$([p1] Ψ). This implies that for all v with (y,v) $\in m_{\mathcal{Z}}$(p1) we have v $\in m_{\mathcal{Z}}(\Psi)$. We can conclude (y,v) $\in m_{\mathcal{Z}}$(if b then p1 else p2 fi) so that u $\in m_{\mathcal{Z}}$([p; if b then p1 else p2 fi] Ψ) follows. The case y $\notin m_{\mathcal{Z}}$(b), i.e., y $\in m_{\mathcal{Z}}$($\neg$b) is shown analogously. □

Loop: To prove the soundness of the loop rule we will have to show that,

If $\Gamma \models$ [p] φ,
Γ, $\varphi \wedge$ b $\models$ [p1] φ, and
$\Gamma \models \varphi \wedge \neg$ b $\rightarrow \Psi$, then the following holds:

$\Gamma \models$ [p; while b do p1 od] Ψ.

Proof: Suppose u $\in m_{\mathcal{Z}}(\beta)$ for all $\beta \in \Gamma$, and v is a valuation so that (u,v) $\in m_{\mathcal{Z}}$(p). If there is an n $\geq$ 0 and $v_0, \ldots, v_n$ with v = v_0 and $v_i \in m_{\mathcal{Z}}$(b) for 0 $\leq$ i < n and (v_i, v_{i+1}) $\in m_{\mathcal{Z}}$(p1) and $v_n \notin m_{\mathcal{Z}}$(b), then we have to show that $v_n \in m_{\mathcal{Z}}(\psi)$. In order to do so, we have to show by induction on i < n that $v_i \in m_{\mathcal{Z}}(\varphi)$. For i = 0, this is trivial since v_0 = v and our first assumption $\Gamma \models$ [p] φ. In the induction step

we have $v_i \in m_{\mathcal{Z}}(\mathrm{b})$ and by the induction hypothesis $v_i \in m_{\mathcal{Z}}(\varphi)$. From the second assumption we get $v_{i+1} \in m_{\mathcal{Z}}(\varphi)$ because $(v_i, v_{i+1}) \in m_{\mathcal{Z}}(p_1)$ finishing the induction. This shows $v_n \in m_{\mathcal{Z}}(\varphi)$ so that $v_n \in m_{\mathcal{Z}}(\varphi \wedge \neg \mathrm{b})$ follows. The third assumption shows $v_n \in m_{\mathcal{Z}}(\psi)$. □

3.3 Incompleteness of this system

Semantic completeness of a formal deductive system is the converse of soundness of that system. If all of the tautologies of a formal system are theorems, then we can call that formal system *semantically complete*. That is, for a formal system S is to be semantically complete, the following property should be satisfied:

$$\models_S \varphi \rightarrow \vdash_S \varphi$$

In [5], Harel presented some simple axiom systems (for various programming languages) which are arithmetically complete. This means that, DL formulas can be effectively converted to a set of predicate logic (PL) formulas using these systems. The generated PL-formulas can be considered as verification conditions and have to be proved using first-order reasoning. These systems use Hilbert-style axiomatization which involves a huge number of axioms and very few inference rules. This feature makes these systems almost impossible to be implemented for practical applications [6].

For the following two reasons our system cannot be complete:

- Implementation of the Hoare logic rules for dynamic reasoning.
- Choice of the model: The set of integer numbers.

As the main focus of this system is to be used as an educational tool, we have implemented only the rules of Hoare logic for the partial correctness assertion. It was not necessary for us to implement the rule for the total correctness assertion because, the implementation of the Hoare calculus was sufficient for ensuring the educational use of this system. But, this feature makes this system incomplete. To make this system complete, the rule for the total correctness assertion must have to be implemented.

The other reason for which this system is incomplete is the choice of the implicit model, namely the set of the integer numbers, which is incomplete. Gödel's incompleteness theorem states that any effective first-order theories which include a suffi-

cient portion of the theory of the natural numbers can never be both consistent and complete. This is a theorem about an arithmetic structure, where the implicit model is the natural numbers. According to this theorem, any arithmetic that can model these natural numbers suffers from the problem of incompleteness [22, 23]. As we have used a first-order arithmetic model, the set of integer numbers, it is not possible to have completeness in this deductive system. One possible solution to this problem could be the use of the set of real numbers as the implicit model instead of the set of integer numbers. But as we mentioned above, due to the absence of a rule for the total correctness assertion, it would not be possible for this system to be complete even if we use any model which is complete.

Chapter 4

Overview of the System

In the first section of this chapter, we will discuss the effectiveness of this system as an educational tool. To derive every step of a proof in this system, the user will have to interact with the system all the time. With the help of user-friendly GUI and step-wise user interactions, a learner of program verification techniques can easily get the idea of the methods of deriving a proof.

4.1 Demonstration of the system

This system accepts formulas of all possible forms discussed in Section 2.3.1. Figure 4.1 shows the result of entering a valid and invalid formula in this system.

Suppose, the valid formula shown in the above example is entered into the system. As we can see from Figure 4.2, the assertion window and the proof explorer window are updated accordingly with the entered formula text.

The button corresponds to implication rule is activated because this is an instance of implication formula. Entering that button results the assumption window and assertion window to be updated with the appropriate text (Figure 4.3).

As the last statement of the program in the entered formula is an while statement, the button corresponds to the application of loop rule gets activated. Entering this button results the system to prompt the user to enter an invariant (Figure 4.4).

As we can see from the proof explorer window of Figure 4.5, three proof obligations are generated as the children of the while statement after taking the invariant from

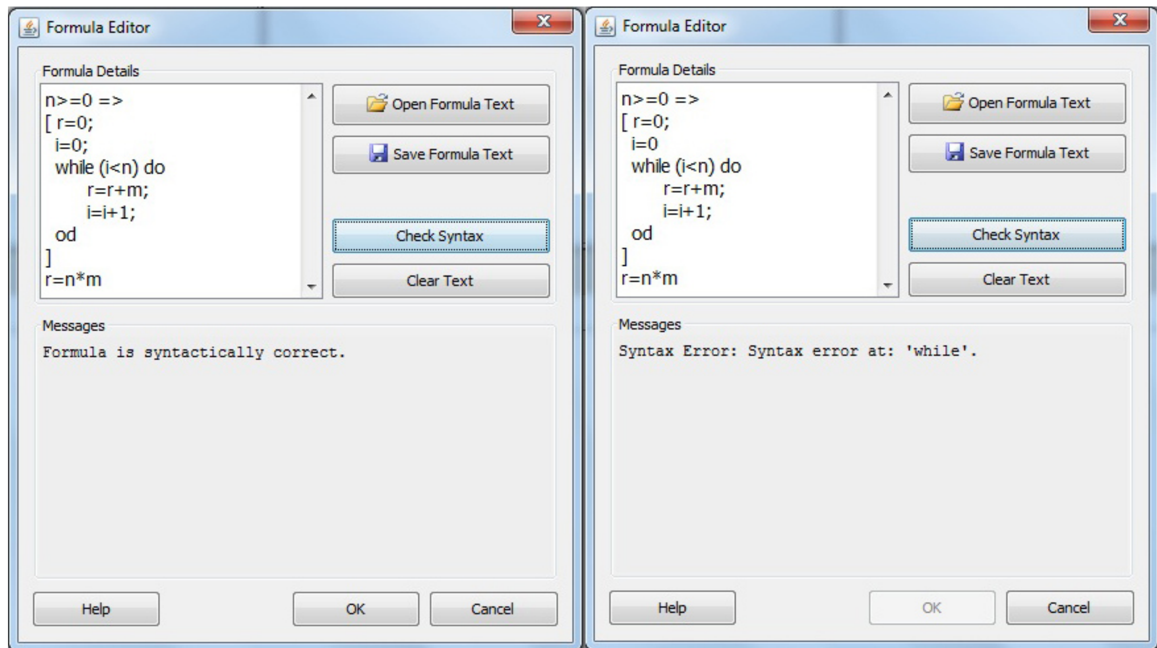

Figure 4.1: Example of Valid and Invalid formulas

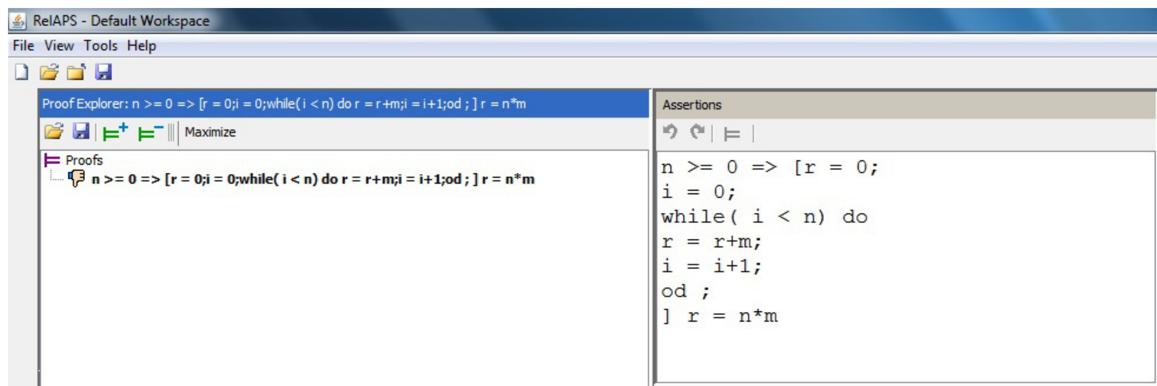

Figure 4.2: A valid formula is entered

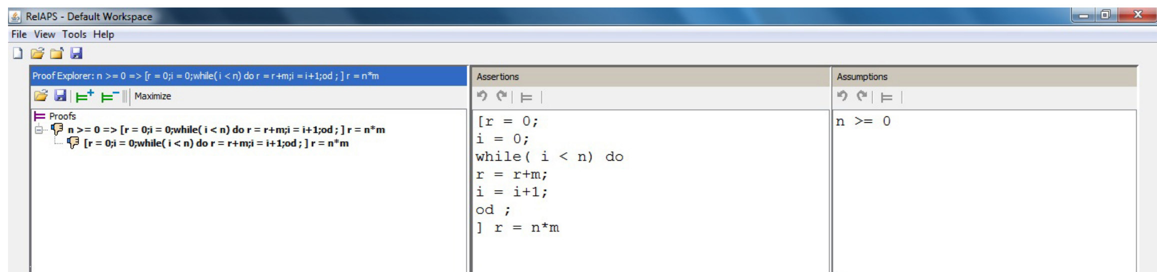

Figure 4.3: After application of the implication rule

user. The assumption and the assertion window are also been changed with the updated data.

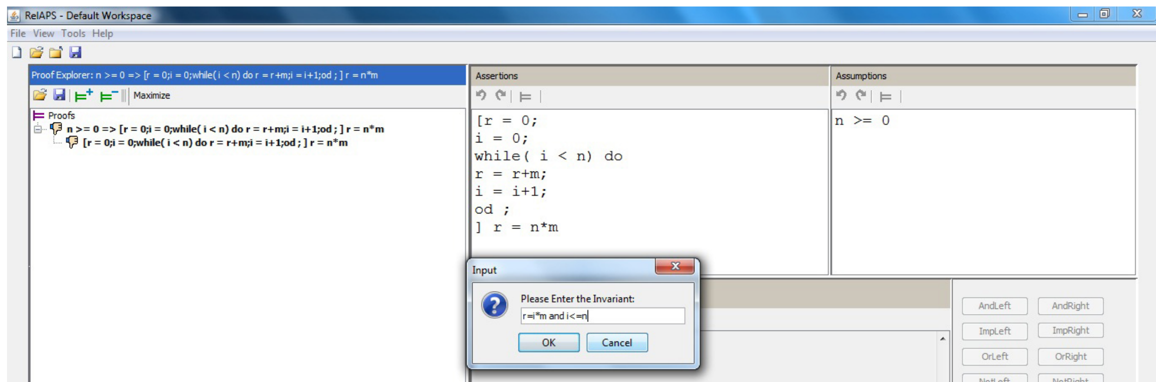

Figure 4.4: User is asked to enter the loop invariant

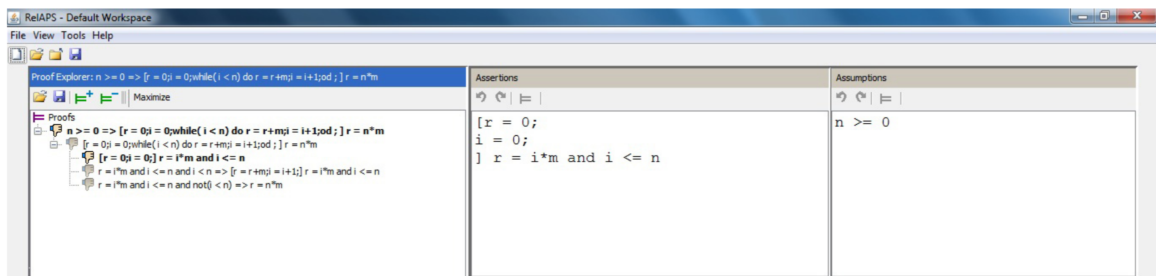

Figure 4.5: While rule has been applied

For the purpose of equational reasoning, users are allowed to select a formula or term from the assertion or assumption window and drag it down to the derivation window. In derivation window, depending on the selection of user, a list of axioms is loaded. User can apply an appropriate axiom from the list to get the desired derivation. An example of the application of axiom rules is shown in Figure 4.6.

Once the derivation has been finished in derivation window, user can replace the selected term or formula in assertion or assumption window with the derived term or formula. As we can see from the Figure 4.7, the identity relation in the assertion window is trivial, which proves the validity of this portion of this proof. If the modified formula in the assertion window is the same as one of the current assumptions, or if it is trivial, the system will recognize this part of the proof as a valid proof. This is reflected in the proof explorer window with a tick mark shown to the left of the proved portion (Figure 4.8). This feature of highlighting a sub-proof will confirm the user of this system that they are in the right track of deriving a proof.

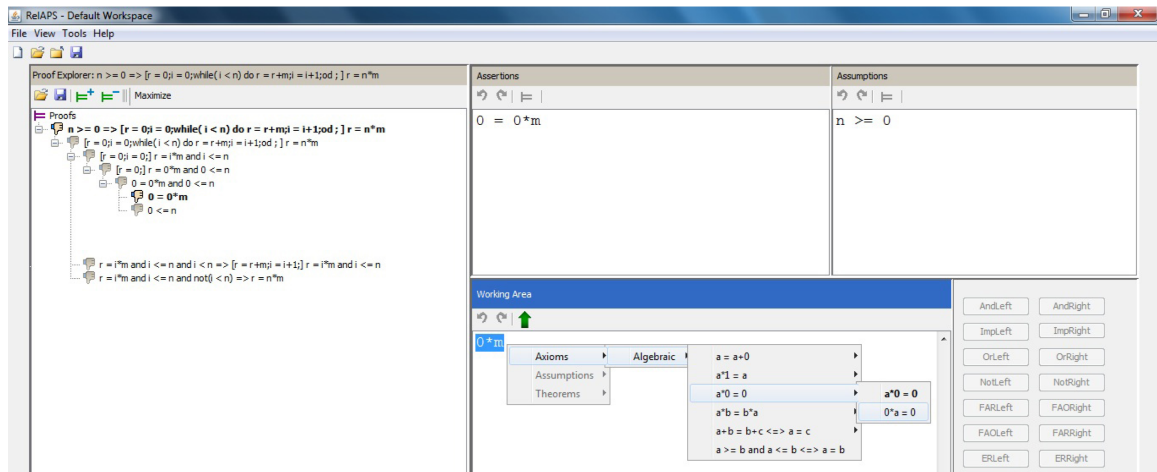

Figure 4.6: List of axioms is loaded

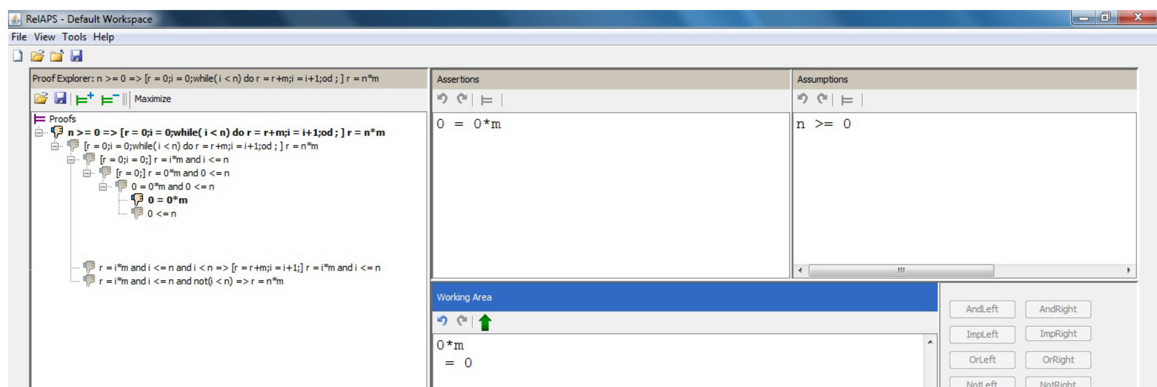

Figure 4.7: Application of user selected axiom

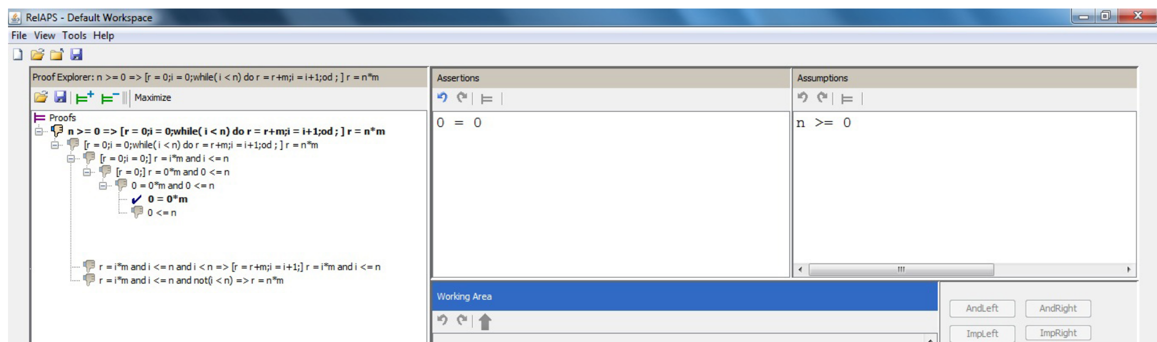

Figure 4.8: A portion of the proof is done

Finally, when the proof is done for an entered formula, a completed proof is shown with a larger sized tick mark in front of the root of the tree in the proof explorer window. This feature confirms the user about the completion of the proof. An example of a completed proof is illustrated in Figure 4.9.

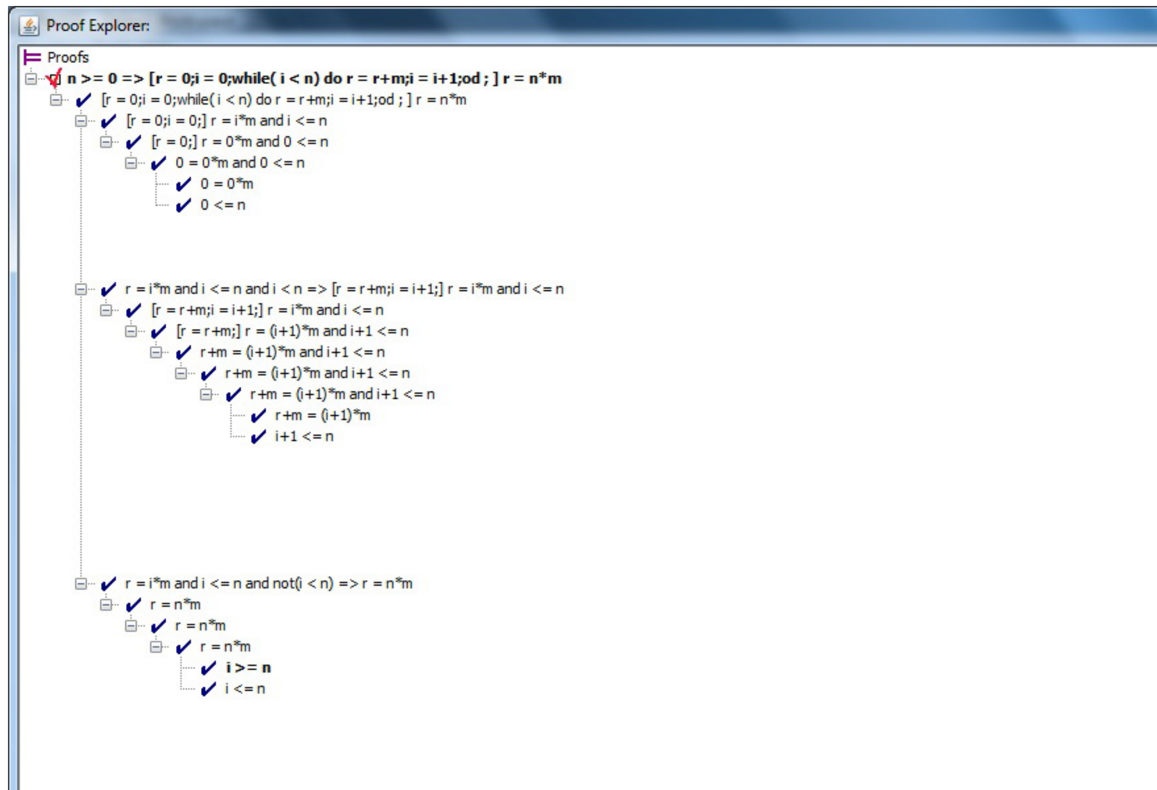

Figure 4.9: A completed proof

4.2 Demonstration of the inference rules

In this section, we demonstrate the use of each of the inference rules separately with the help of screenshots.

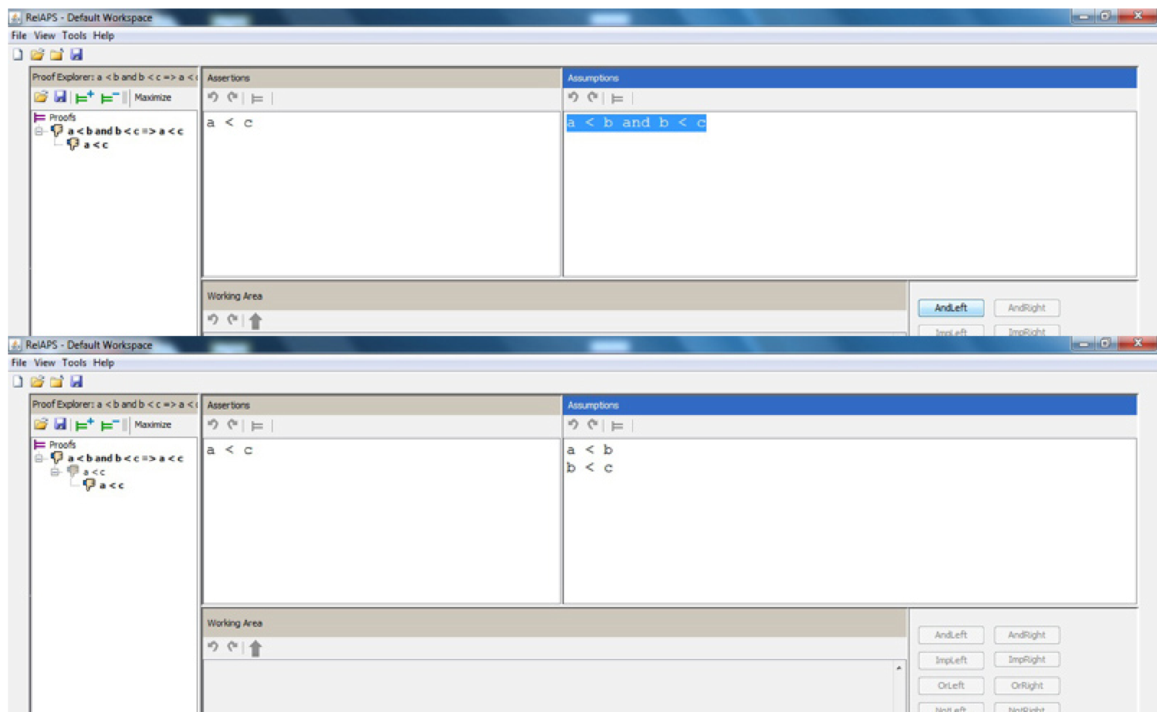

Figure 4.10: Application of the AndLeft rule

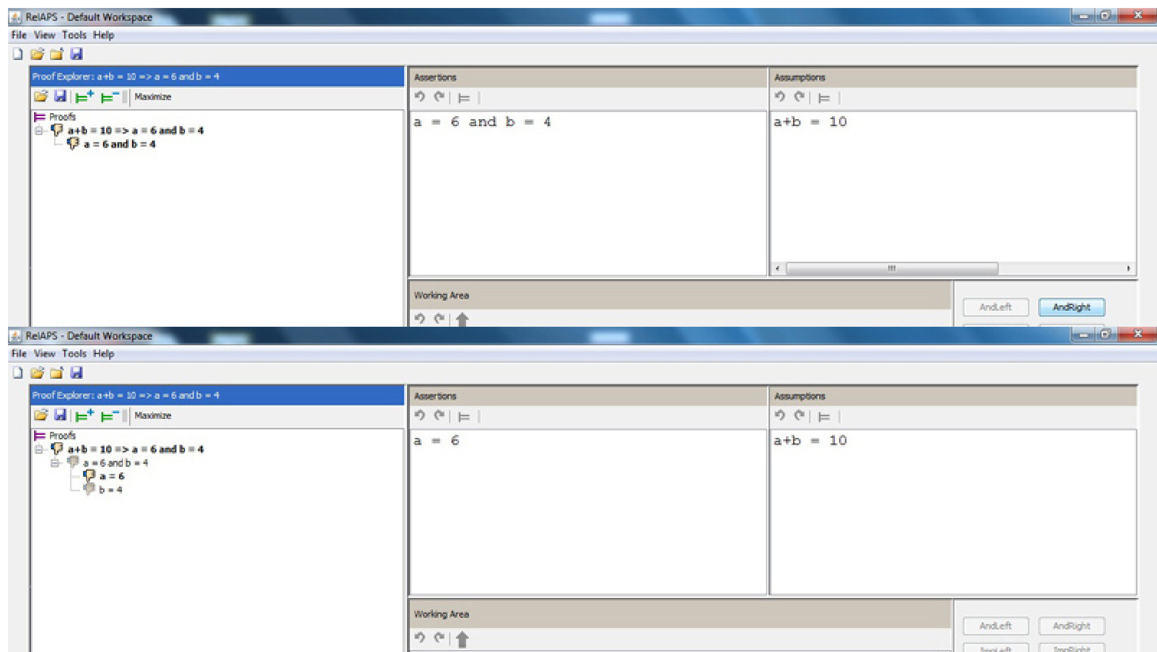

Figure 4.11: Application of the AndRight rule

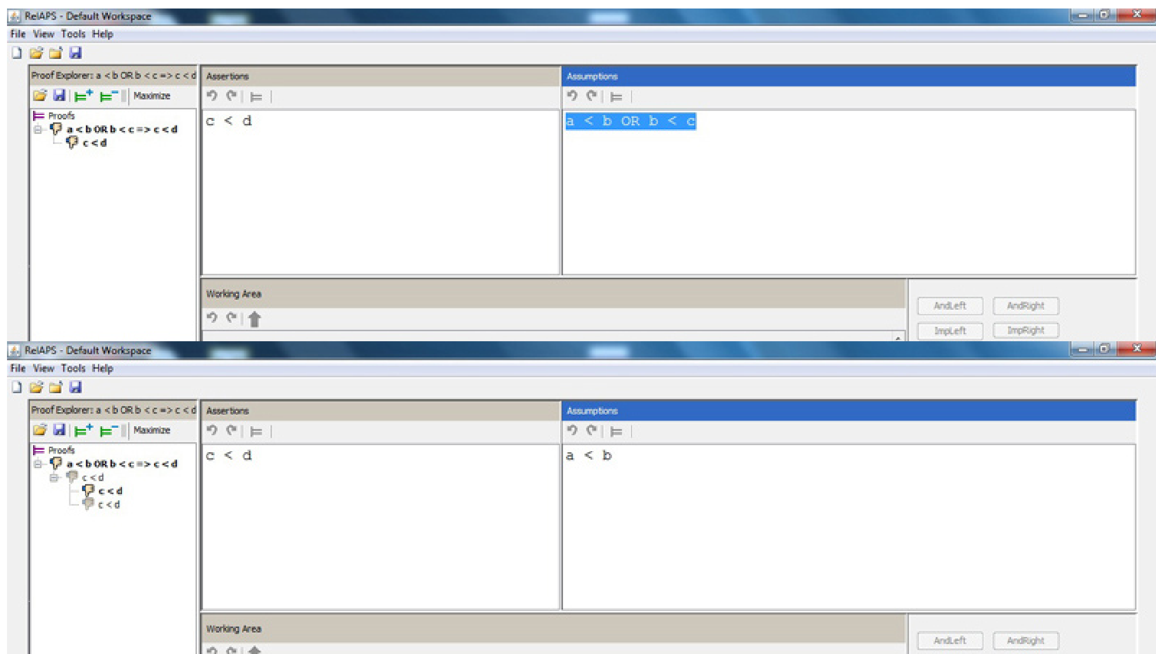

Figure 4.12: Application of the OrLeft rule

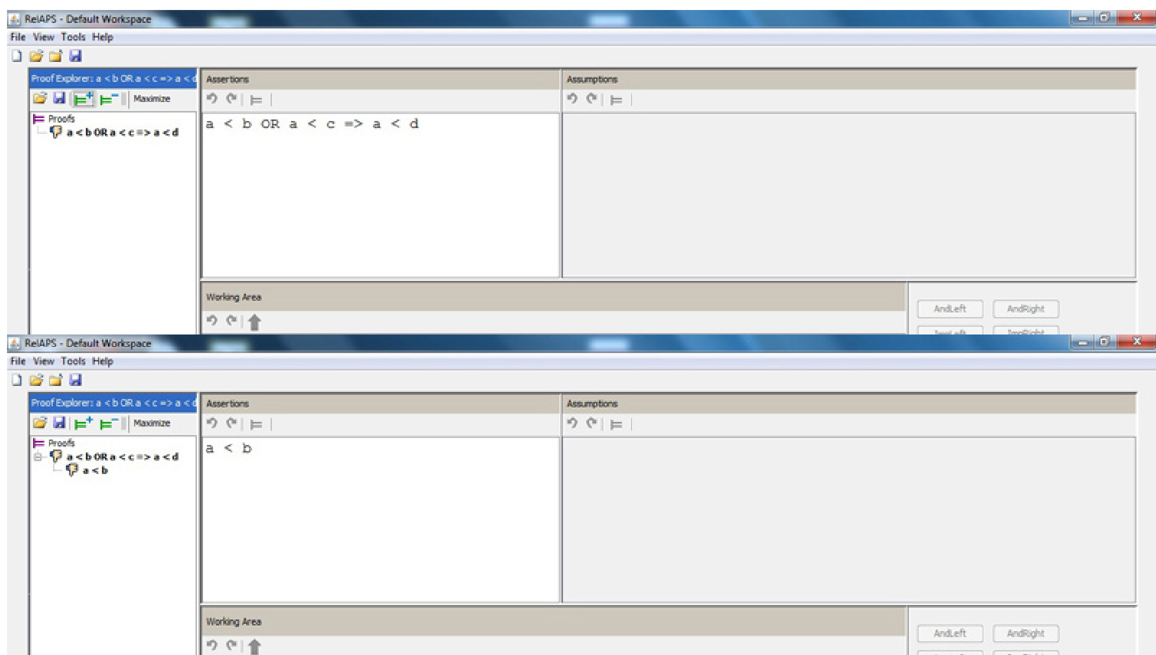

Figure 4.13: Application of the OrRight rule

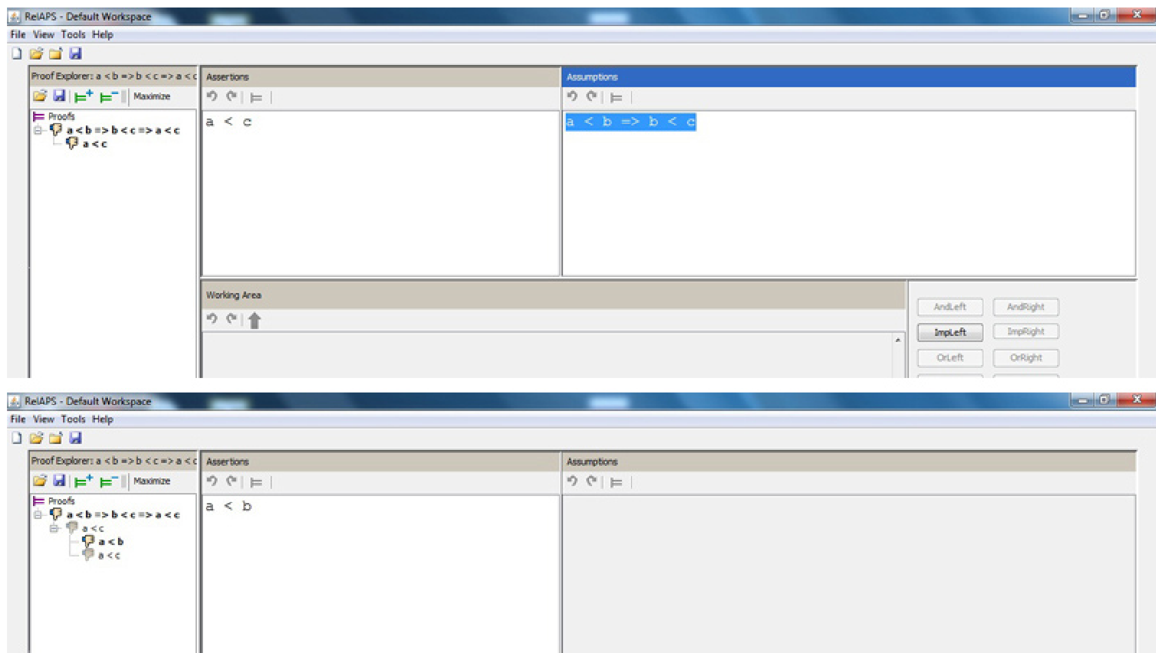

Figure 4.14: Application of the ImpLeft rule

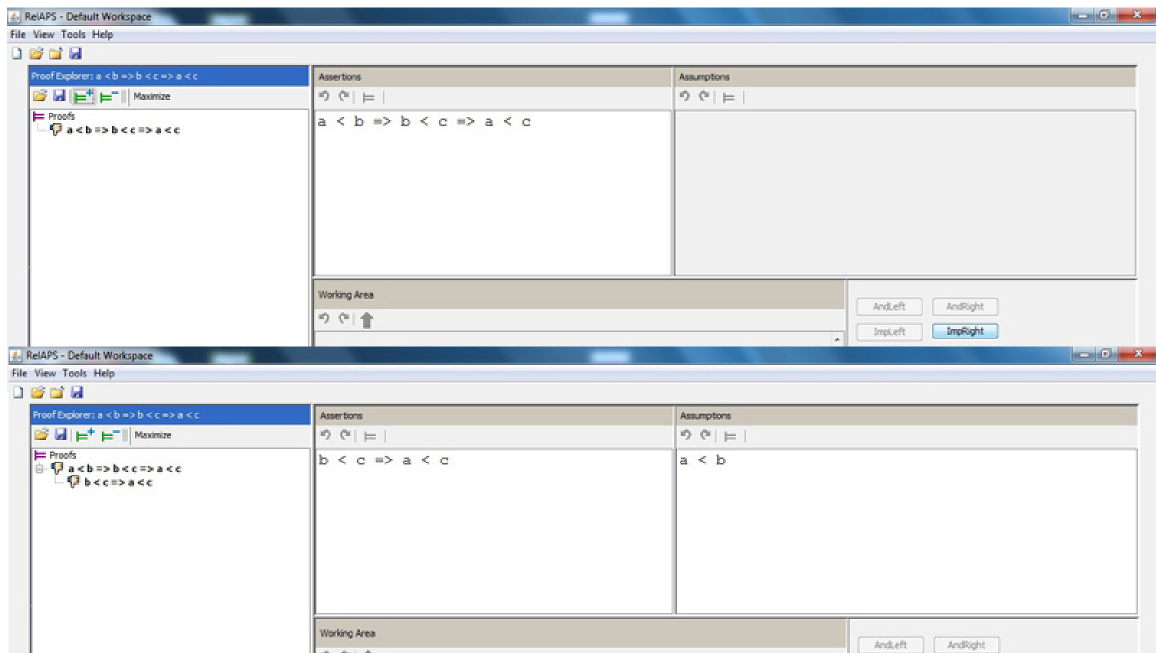

Figure 4.15: Application of the ImpRight rule

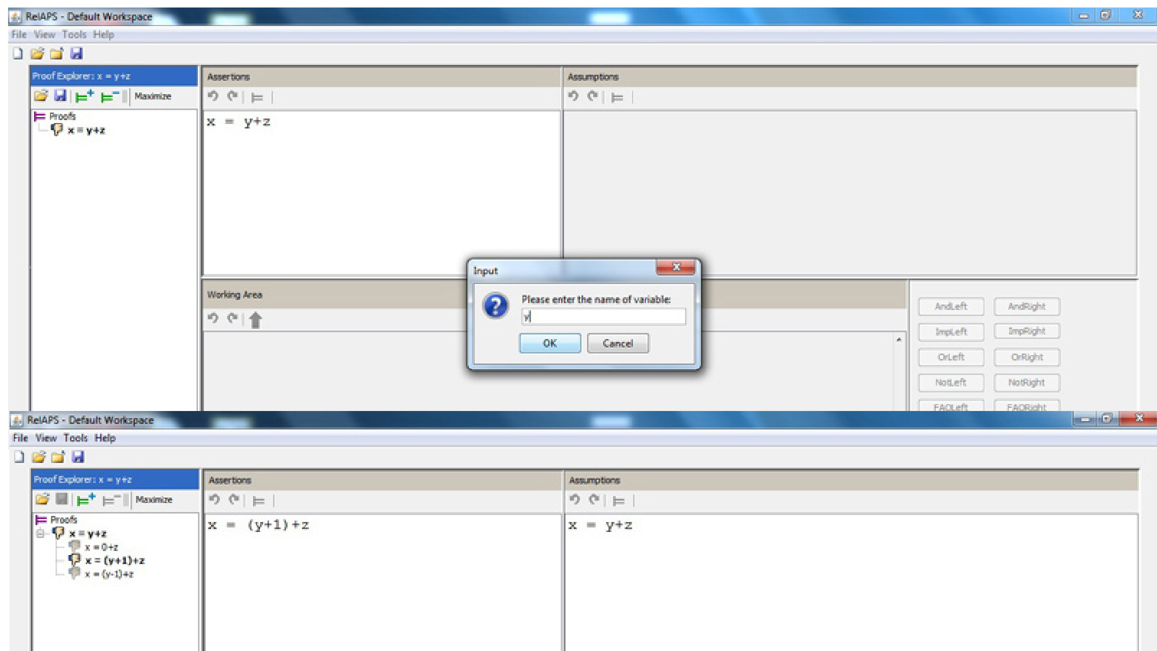

Figure 4.16: Application of the Induction rule

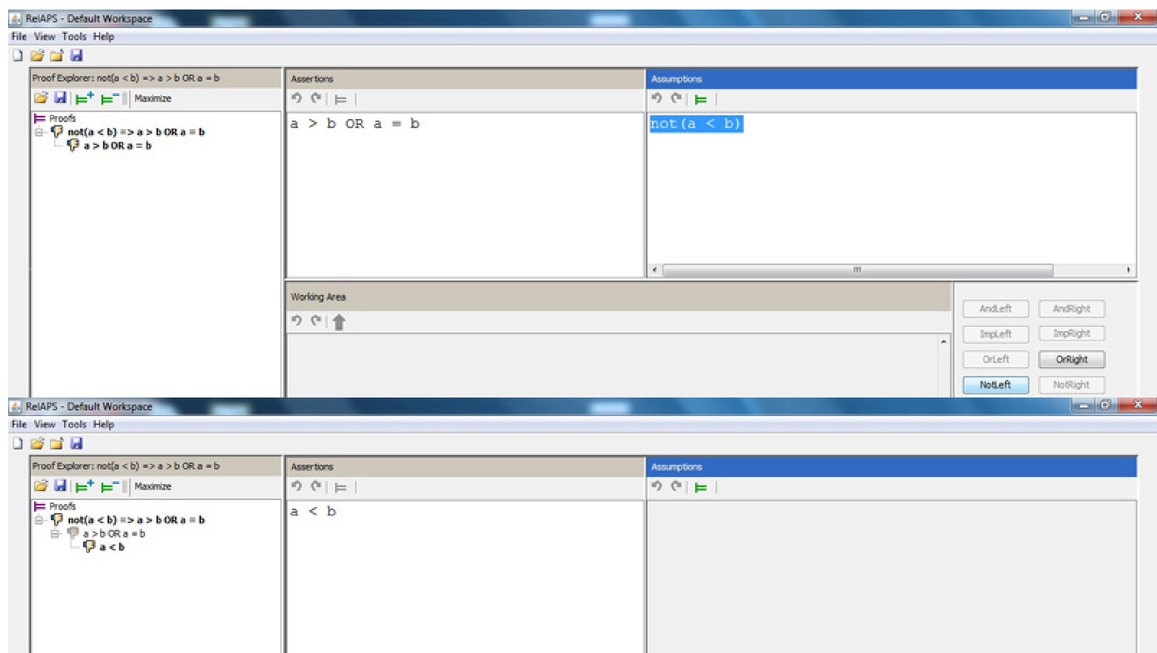

Figure 4.17: Application of the NotLeft rule

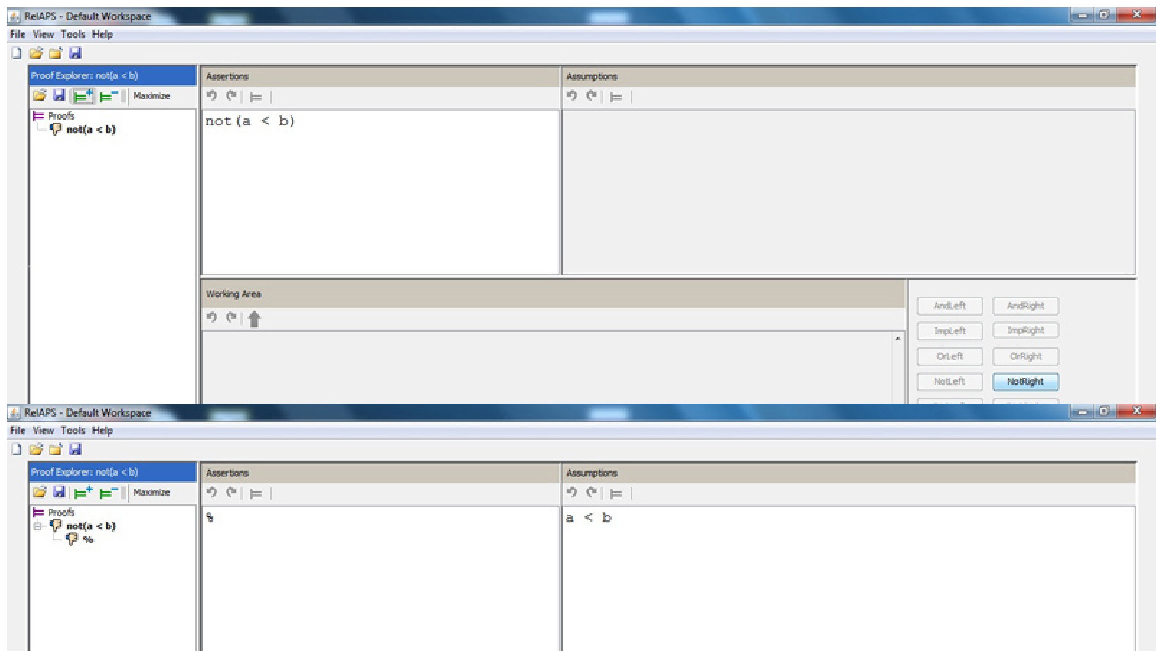

Figure 4.18: Application of the NotRight rule

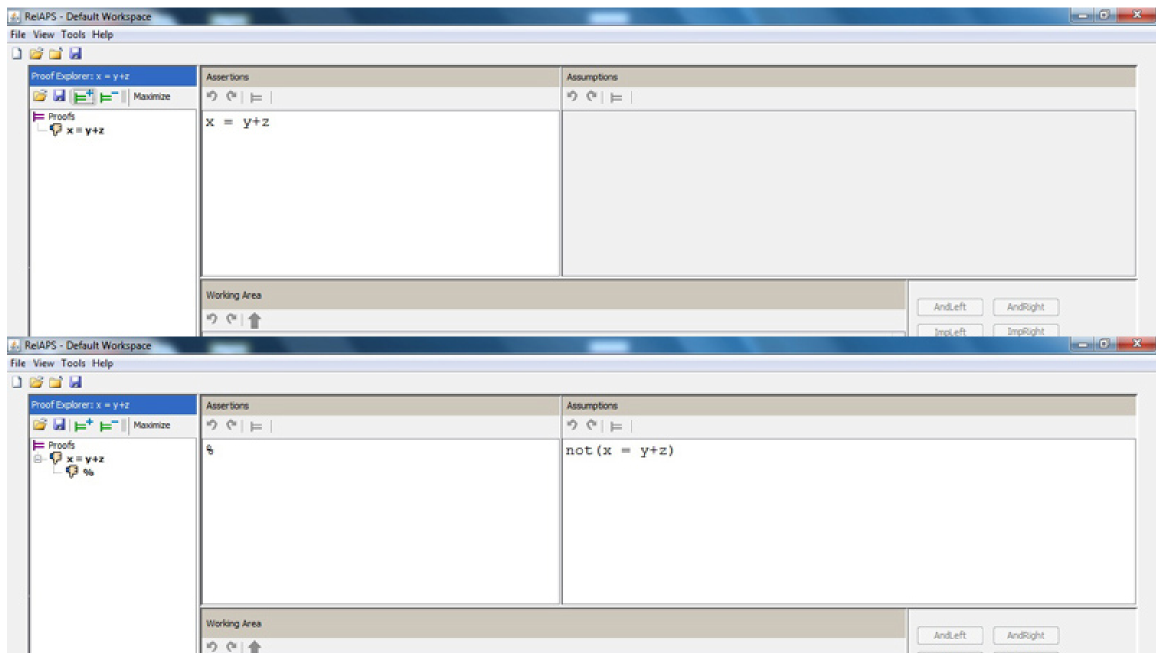

Figure 4.19: Application of the PBC rule

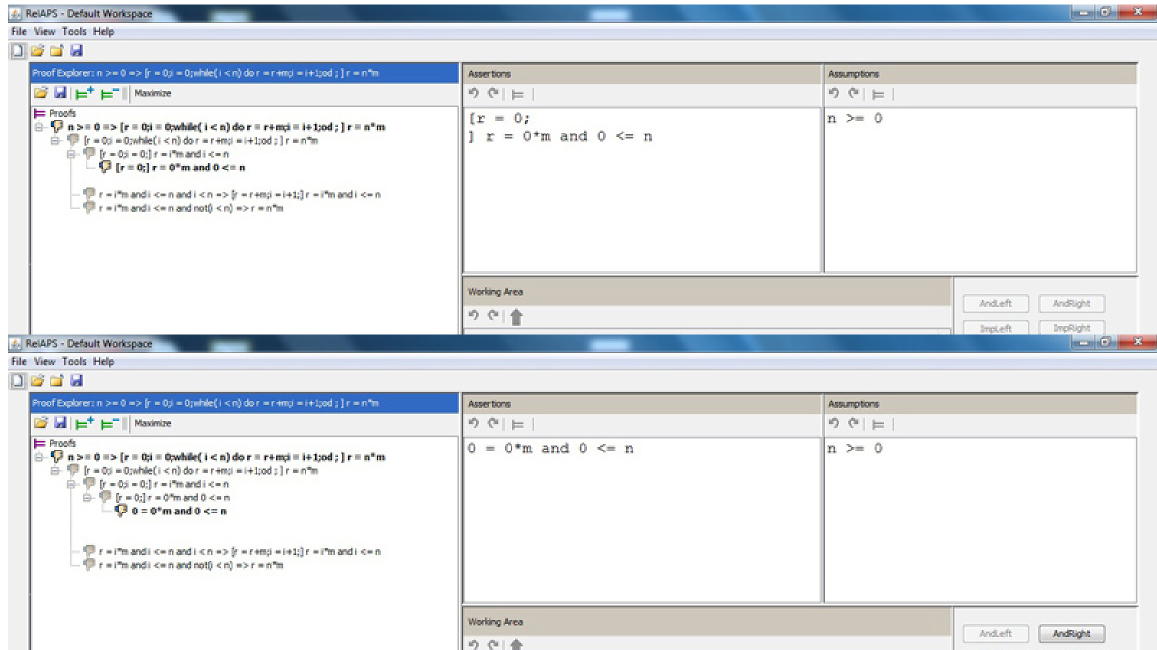

Figure 4.20: Application of the Assignment rule

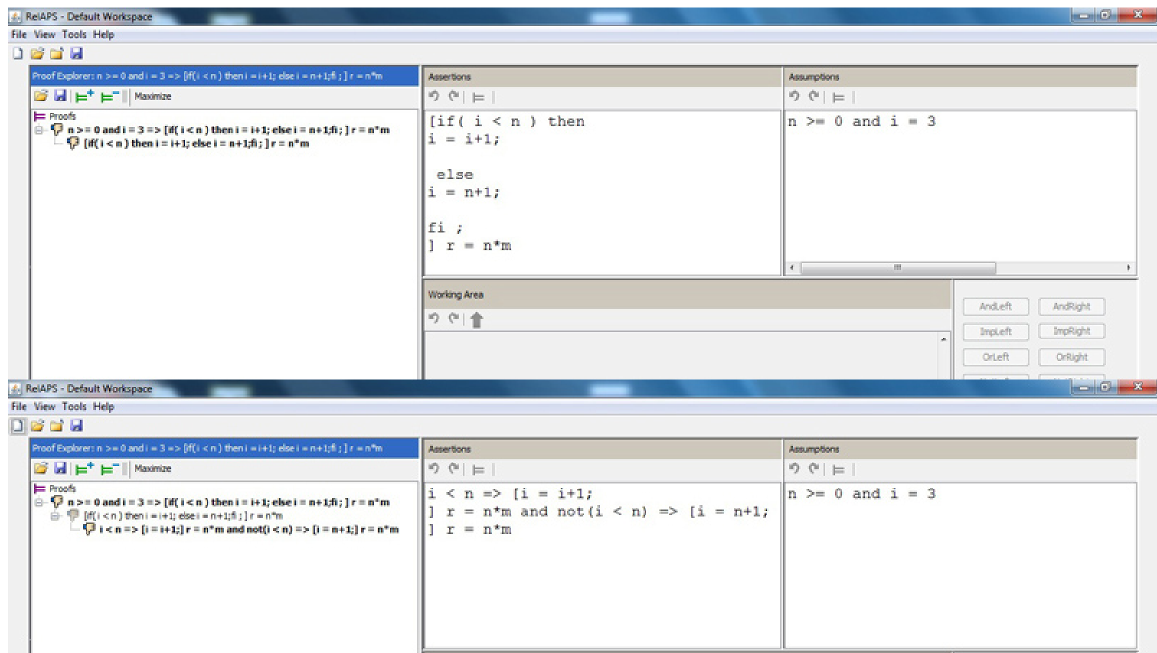

Figure 4.21: Application of the if-else rule

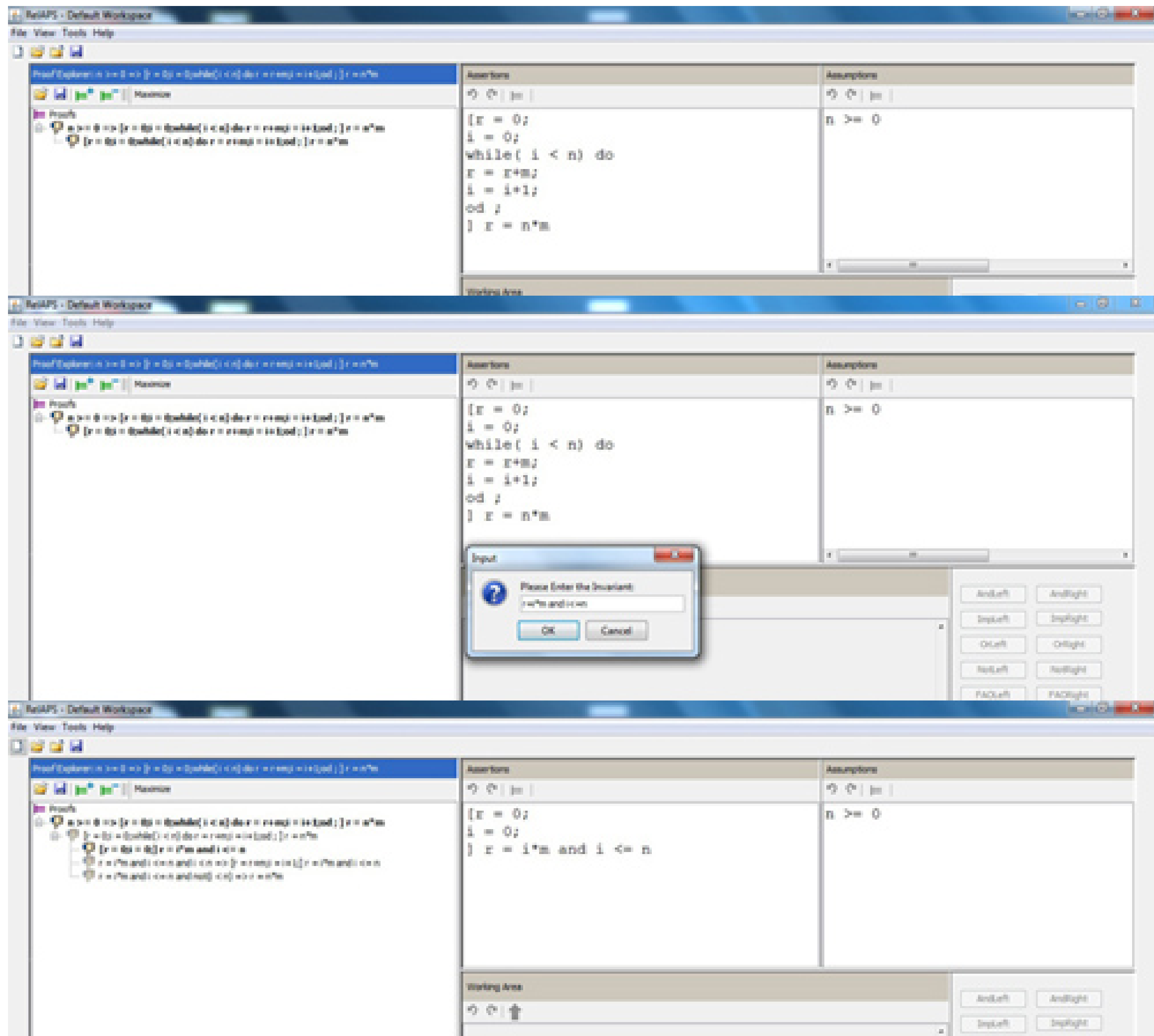

Figure 4.22: Application of the loop rule

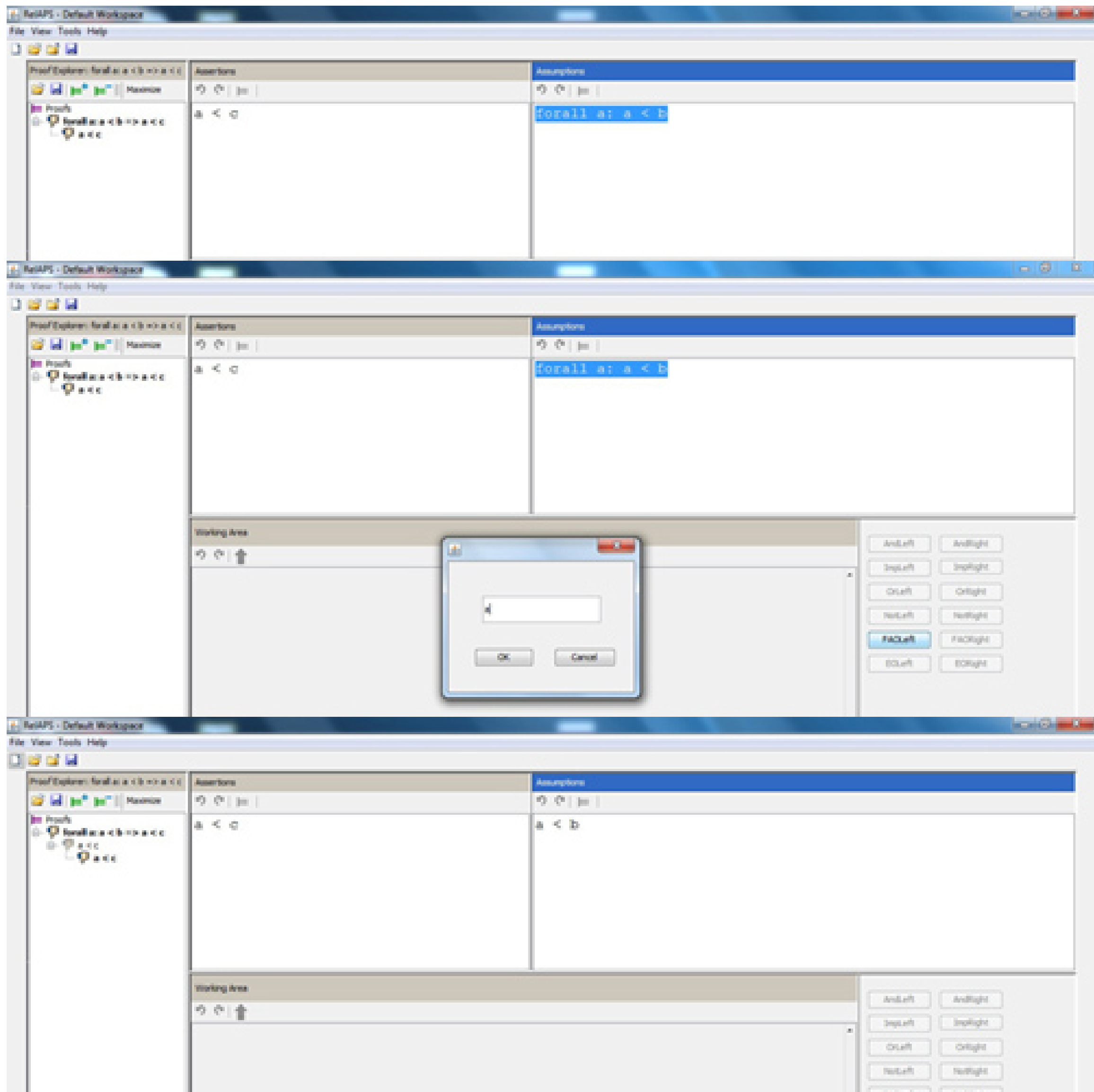

Figure 4.23: Application of the ForallLeft rule

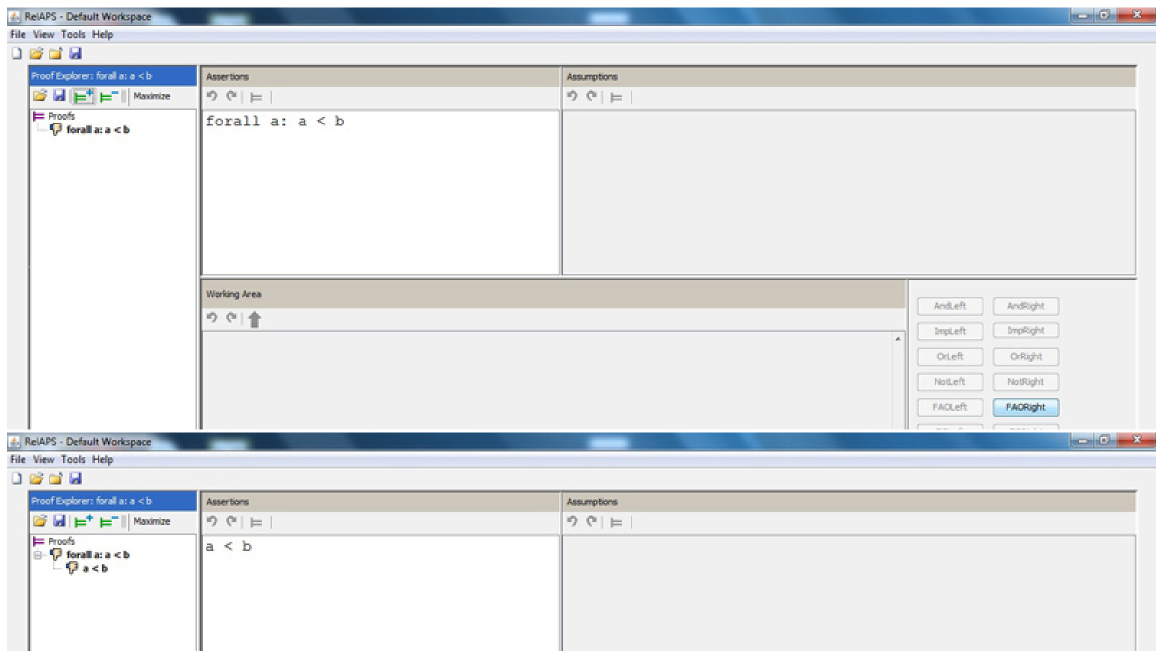

Figure 4.24: Application of the ForallRight rule

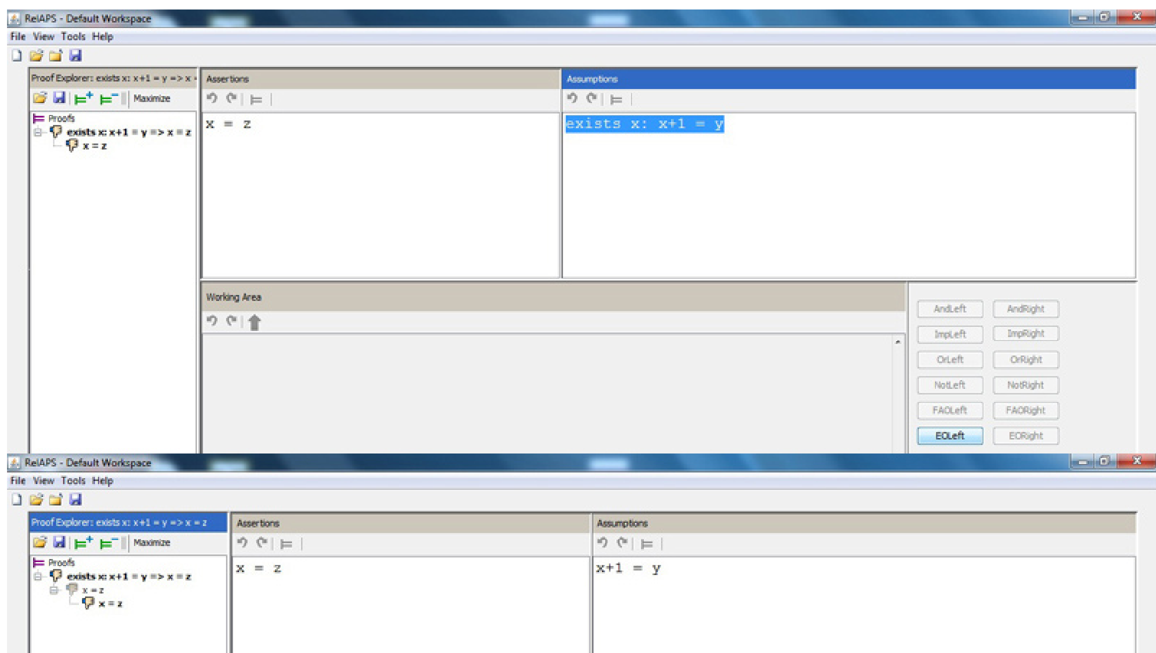

Figure 4.25: Application of the ExistsLeft rule

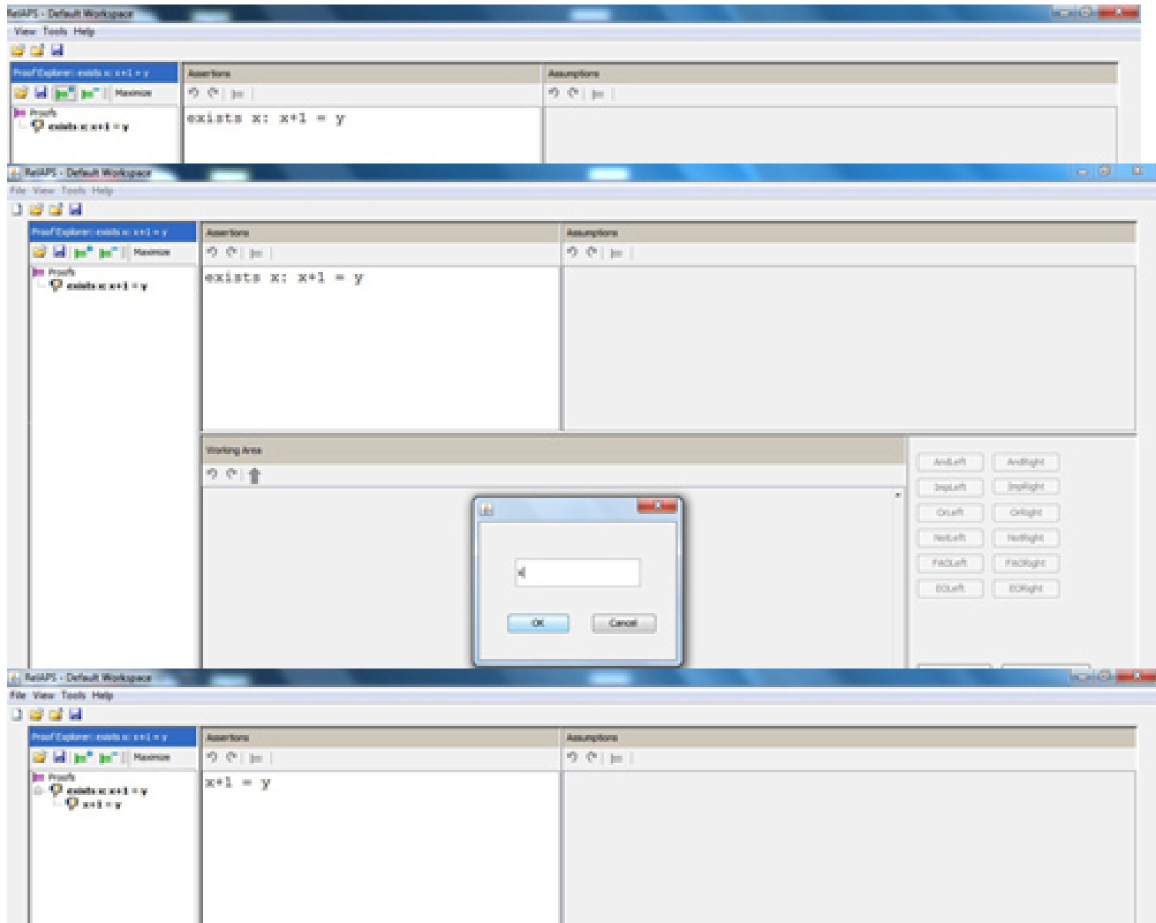

Figure 4.26: Application of the ExistsRight rule

Chapter 5

Implementation

In this chapter, we will briefly discuss the methods of implementing the system. The class hierarchies and relationship between some important classes will also be discussed here.

5.1 Brief explanation of source code

This deductive system for FODL has been implemented in the platform-independent programming language Java using the Netbeans Interactive Development Environment. Precisely, Version 1.7.0 of the Java Development Kit and Version 7.1.1 of NetBeans IDE were used. As the set of first-order formulas is a subset of the set of dynamic logic formulas, one part of this system required us to implement the rules of classical first-order logic. But Bahar Aameri has already developed those rules of first-order logic in her MSc Thesis [1]. RelAPS is an interactive system developed by J. Glanfield which can be used to perform a relation-algebraic proof similar to doing it using pencil and paper [8]. Bahar extended this system to full first order logic. So we have reused her code with some modification to adapt with our system. But as the main part of our work involves the implementation of the rules of Hoare logic for the reasoning about programs, we had to create some new classes in addition to the modification of existing classes. The major changes in the code of that system are summerized below.

A new package named **DL** has been added which contains some classes to support different types of program statements. For each type of these program statements, a new button has been added to the GUI. For Interacting to each of these buttons, we had to make some changes in the classes of **GUI** package, and some new classes has

been developed in the **Rules** package. In the RelAPS system, users are allowed to select a specific type of theory from a set of choices. Depending on a specific choice, a particular theory, which includes a list of axioms, theorems and a set of operators, is loaded from the file directories. But in our system, we have only one type of theory, which gave us the flexibility to hard code the operation information in the grammar file. This modification in loading operation information leads us to make some major changes in the `OpInfo` class of Operations package. A list of axioms that corresponds to the property of integer numbers is being loaded during the initialization of this system. The combination of JFlex and BYacc/J is used to generate the parser for formulas. JFlex is a very popular tool for generating lexical analyzer in Java. It has built-in support for the parser generator BYacc/J, which is the Java extension to the classical Berkeley Yacc parser generator [19]. We had to make some major changes in the existing grammar file to cope with the requirements of formulas of our system.

In Bahar's system, the only defined predicate symbols are $\langle$, $\rangle$ and =. In the formula package, some new classes has been added namely, `GreaterThanEq` and `LessThanEq`, to support the predicate symbols $\langle=$ and $\rangle=$. In addition, another class named `DLFormula` is also been created in the same package to support a dynamic logic formula that includes a program. A class named `FormulaVar` has been added in the Terms package to take care of variables and constants in the formulas of our system. In addition to that, we had to make some minor changes in some other classes in **Terms** and **Operations** package for satisfying new requirements of finding or replacing variables.

The first step of our implementation was to define a proper language. The language is comprised of a set of terms which includes variables, constants or n-ary function symbols, where a function symbol could be any arithmetic operators like, plus, minus, multiplication or division. The next step involves the arrangement of a suitable interpretation of the entities of the languages. The main work involved in this step is to search for a proper environment and a model and then defining terms value and formulas validity. For this to work, this system loads some arithmetic axioms during initialization and provide the user with the option to select a particular axiom to apply from a list of appropriate axioms in the derivation window. User of this system can select a formula or term in the assertion or the assumption window and can drop it down to derivation window. After applying appropriate derivation rules, user can send the corresponding derived formula or term back to the assertion or

assumption window. There was already a class developed in RelAPS system named Unifier to establish an environment for mapping from one term to another. But we had to make some changes in different classes of **GUI**, **Formulas** and **Operations** package to adjust the unification operations with our requirements.

The classes in the **ProofFactory** package are used to control the application of the rules and the **GUI** package is used to make necessary changes in the interface of the application. To control all instances of the rules of program reasoning, namely, `WhileRule`, `IfRule` and `AssignmentRule`, a class named `DLProof` is created in **Proof-Factory** package. This package also contains one class for each of the derivation rules. In GUI, for every derivation rule, we have a button, which get activated depending on the context of a working formula. Now, for example, we have a program in the formula and the last statement of that program is an if statement. In this context, the button corresponds to the if rule will get activated. Entering that button will first create a new proof using a static method *createProofs* in `ProofStyle` class and then using *ApplyRule* method of `IFRule` class, one proof obligation will be generated and passed to the **GUI** package. The classes in the GUI package will then take care of updating the tree view of Proof Explorer section, and Assertions and Assumptions window. If the system detects any while statement as the last statement of a program, the user will be asked to enter a suitable invariant. The system will then pass the invariant and the postcondition to the `WhileRule` class, which will eventually generate three proof obligations from these inputs and return those proof obligations back to `GUI` class.

As we are asserting the formulas of this system about every integer number, we were required to implement mathematical induction in our system. For implementing this induction, we have created a new button in GUI. A new class named `InductionRule` has been added to the **Rules** package as well. Clicking on the button corresponds to induction will take a input variable from the user, and then three proof obligations will be generated after applying the induction rule on the working formula.

5.2 Class hierarchies

A formula of our system can take several forms (Figure 5.1). Possible forms of formulas include, an atomic formula, a binary formula or a formula with a program. An atomic formula is a basic formula which could be generated using any predicate

symbols, and a binary formula can be constructed using two atomic formulas. In our implementation a term object is a part of a formula object. A formula object is constructed using one or more number of terms (Figure 5.2).

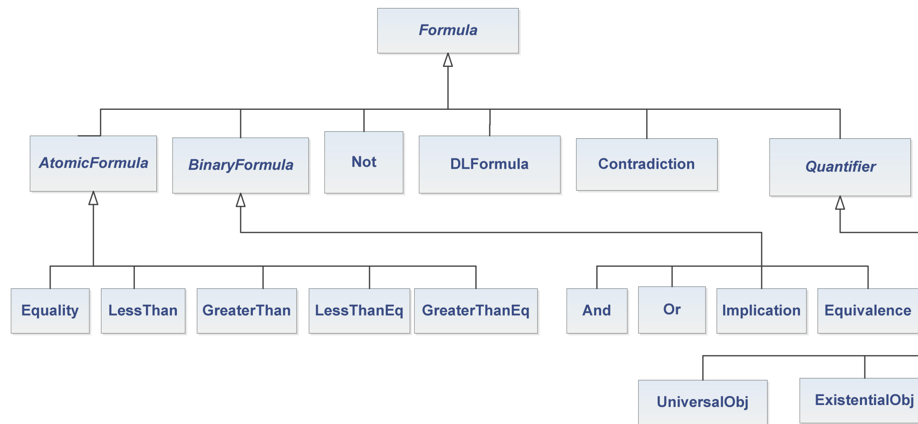

Figure 5.1: Class diagram of Formula

Figure 5.2: Relationship between Formula and Term

If any program statement is detected in the current formula, depending on the type of last statement in that program, an object of `WhileStatement`, `AssignStatement` or `IfStatement` will be generated. Each of these classes are a subclass of an abstract class `Program` (Figure 5.3). This design of our program structure can allow us to utilize the advantages of Java polymorphism.

A term in our system can take three forms: `ObjectVal`, `BinOp` or `FormulaVar`. For each of the quantification variables, an object of `ObjectVal` will be generated. An object of `BinOp` will be instantiated if the system detects any arithmetic operations in the formulas. Finally, for each of the variables and constants in the formulas, a `FormulaVar` object will be created (Figure 5.4).

A `Formula` object and a `Program` object are defined by mutual induction. A `Formula`

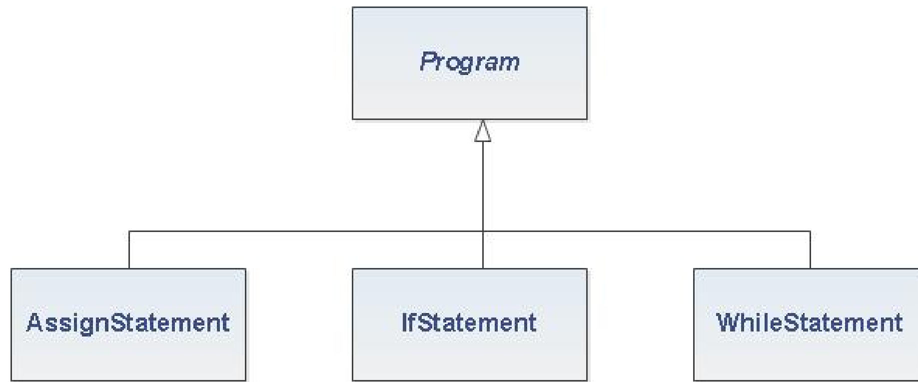

Figure 5.3: Class diagram of Program

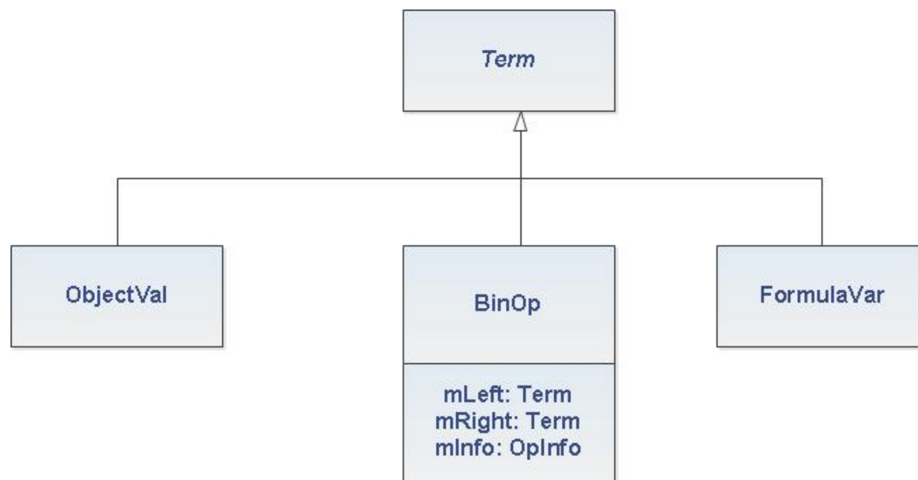

Figure 5.4: Class diagram of Term

object can have any number of programs within it, while a `Program` object can also have any number of formulas within it. In our system, quantifier-free formulas are used as test condition, which is also referred to as *Poor Test* (Figure 5.5).

Figure 5.5: Relationship between Formula and Program

Chapter 6

User Manual

The formulas that can be accepted in this system are defined in Section 2.3. During the proof of a derivation, the users of this system can apply any of the rules mentioned in Section 3.1. As shown in Figure 6.1, a set of buttons have been added to the application interface for this purpose. The buttons are used to apply any of the derivation rules we mentioned in Section 3.1, on the current proof. All derivation buttons are disabled by default except `PBC` and `Apply Induction` buttons. The buttons correspond to the right-hand rules of first-order reasoning are enabled based on a subtree that has been selected in the 'Proof Explorer' window. An appropriate button for left-hand rule of first-order reasoning will be enabled if a user of this system selects an assumption in the 'Assumptions' window. The buttons related to program reasoning will be activated if the formula of the currect proof has a program within it, the last statement of which will determine the activation of a specific button. For example, if we have a program in the current proof, and the last statement of the program an instance of a while statment, then the corresponding button for the application of while rule will be enabled.

The system has two seperate windows namely, `Assertions` and `Assumptions` window, to let the user know about the goals and assumptions of the current proof. The `Assertions` window is used for displaying the assertion of the current proof, which is the right hand side of $\vdash$ in a derivation. The text area of the `Assertions` window is used only to display the current state of the assertion being worked with. The user are allowed to work with only one assertion at a time. This is specified by clicking on the appropriate assertion in the tree view of the `Proof Explorer` window. (Figure 6.2)

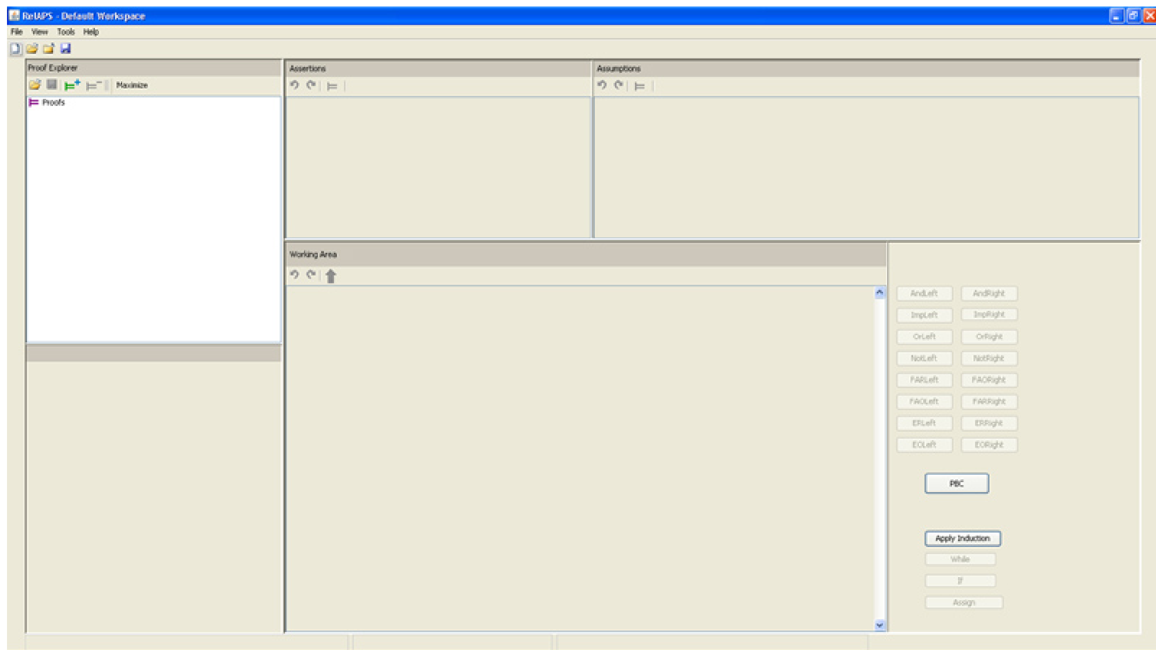

Figure 6.1: The user interface of this system

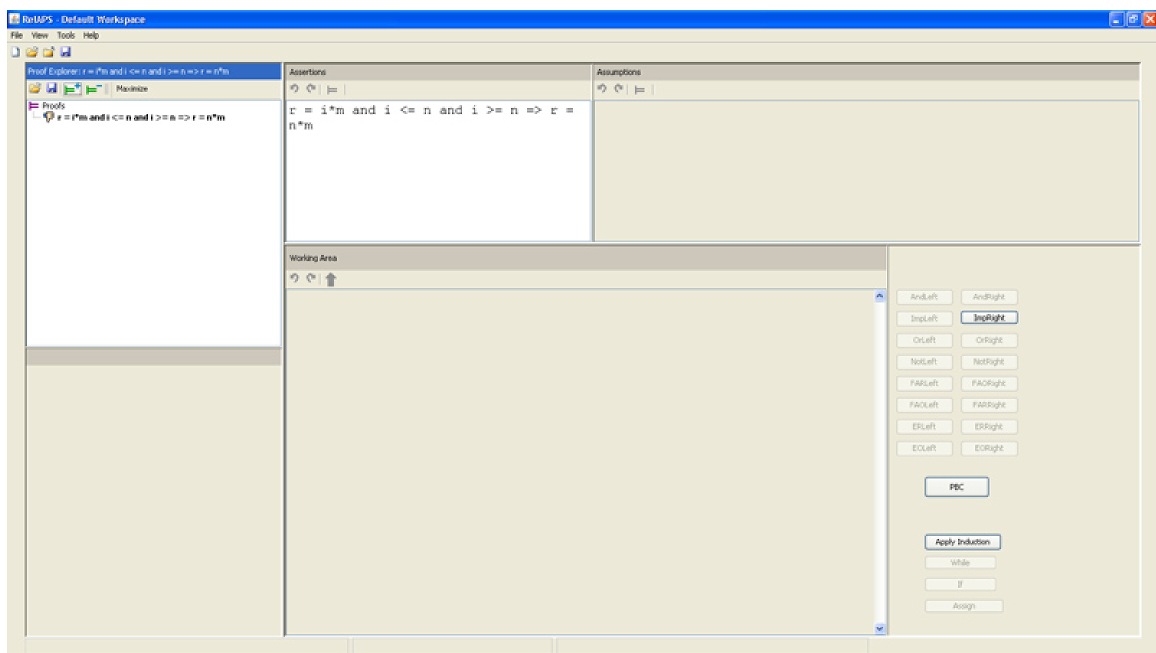

Figure 6.2: The current goal is displayed in the `Assertions` window

The `Assumptions` window is used to display the assumptions that are associated with the current proof. This corresponds to the left-hand side of $\vdash$ in a derivation, i.e., Γ. Figure 6.3 shows the assumptions of the current proof in the `Assumptions` window. A left-hand rule for first-order reasoning can be enabled based on the selection of an

assumtion in the `Assumptions` window. This is demonstrated in Figure 6.4.

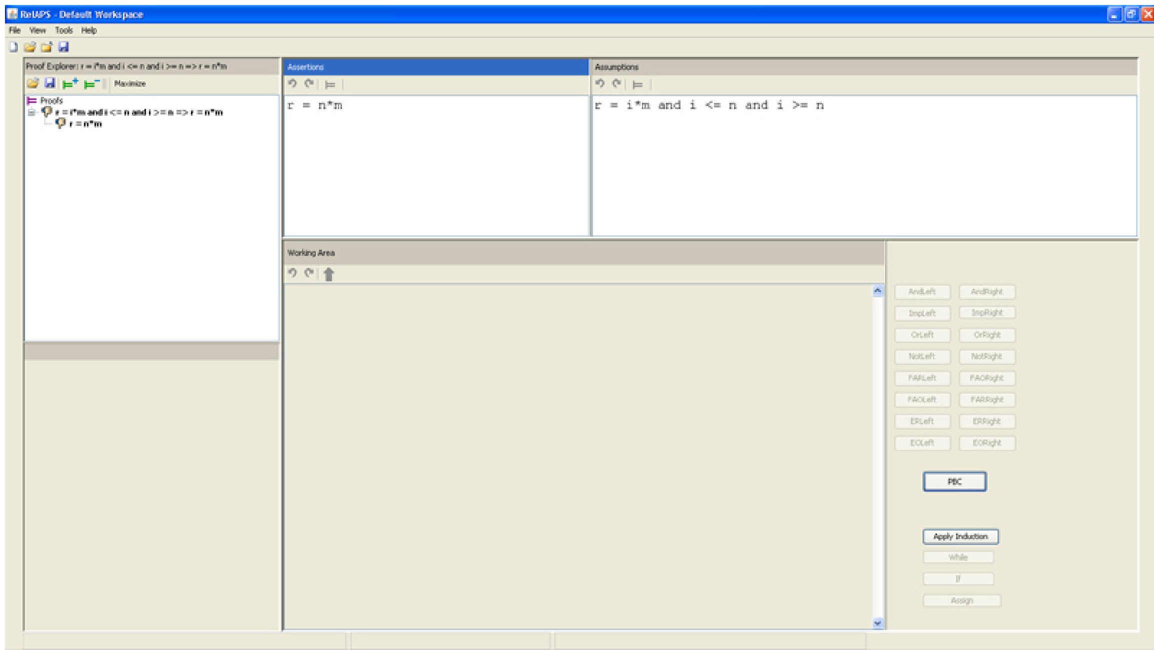

Figure 6.3: The assumption of the current proof is displayed in the `Assumptions` window

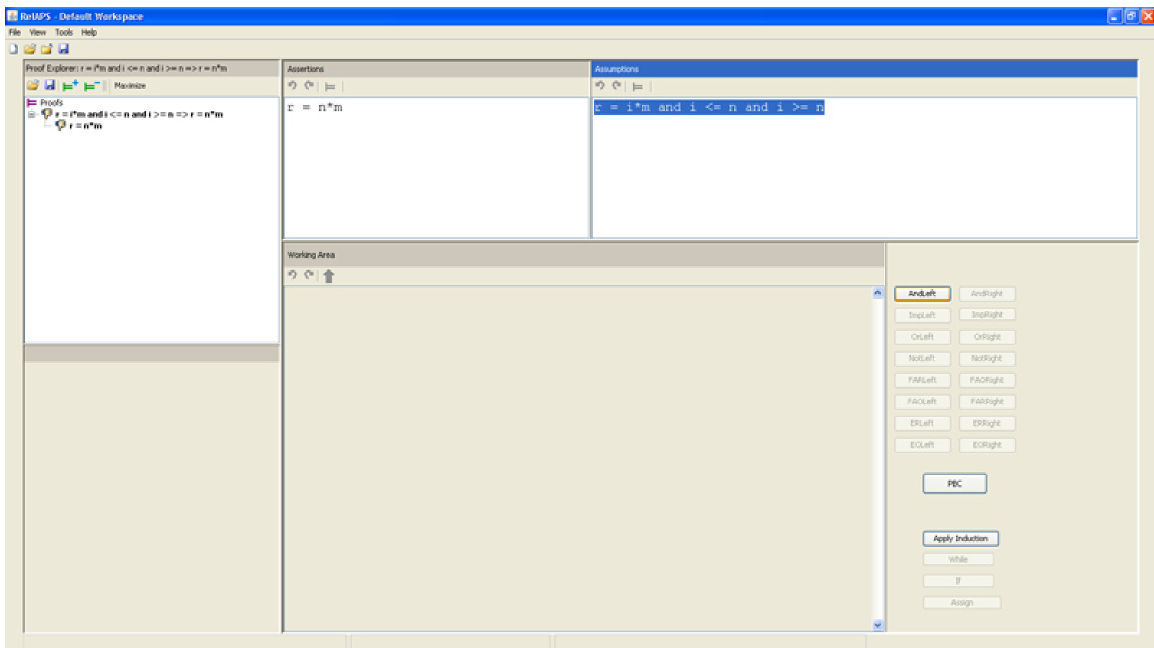

Figure 6.4: Activation of a left-hand rule, AndLeft

The application of this AndLeft rule is shown in Figure 6.5, where we can see multiple assumptions in the `Assumptions` window. As a derivation can have multiple assumptions, a list of assumptions can be placed in the `Assumptions` window. From this list, a user of this system is allowed to select a particular assumption in order to modify it.

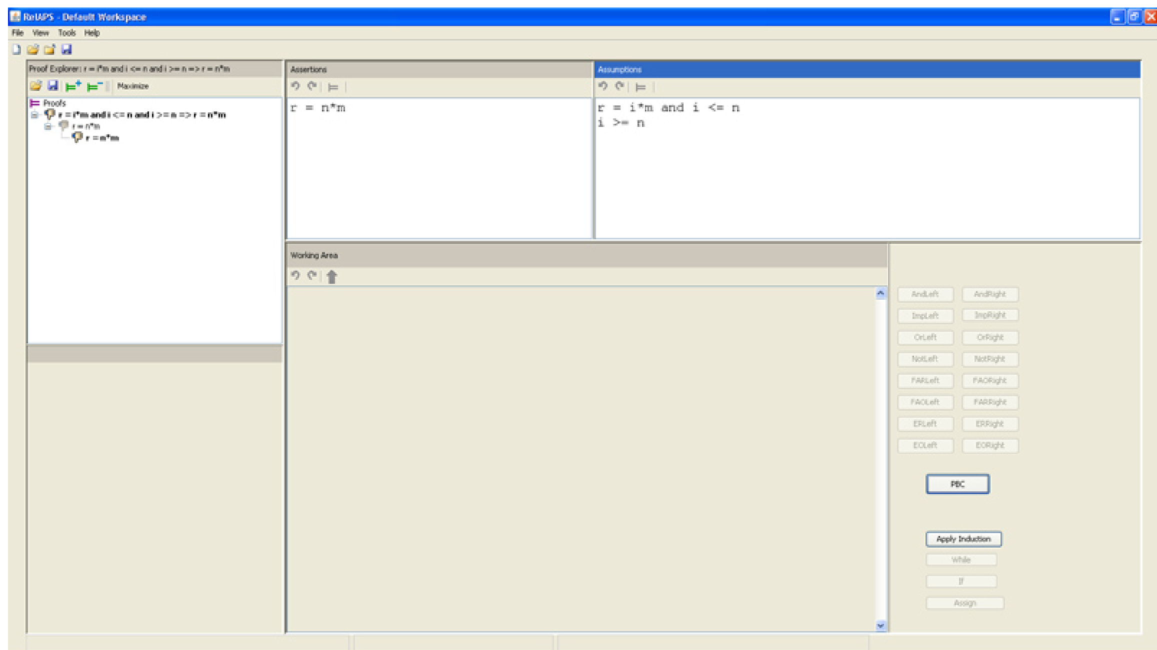

Figure 6.5: The left-hand rule, AndLeft, has been applied

A user has the ability to select a term or formula in the `Assumptions` or `Assertions` window, and drop it down to the `Working Area` window, where an appropriate axiom rule can be applied to the selected term or formula (Figure 6.6). As we can see from Figure 6.6, an atomic formula is selected in the `Assertions` window which can be dropped down to the `Working Area` window by clicking on derivation button in `Assertions` window. Result of clicking on that button is shown in Figure 6.7.

Now, we have one formula $r = n*m$ in the `Assertions` window, and three formulas, $i >= n$, $i <= n$ and $r = i*m$ in the `Assumptions` window. If we select the variable n in `Working Area` window, a list of arithmetic axioms will be loaded. Among them, if we select the axiom as shown in Figure 6.8, a new window will appear. Purpose of this window is to let the user to select a mapping of the unmapped variables. As we can see from Figure 6.9, for the application of the axiom rule $a = b => a >= b$ and

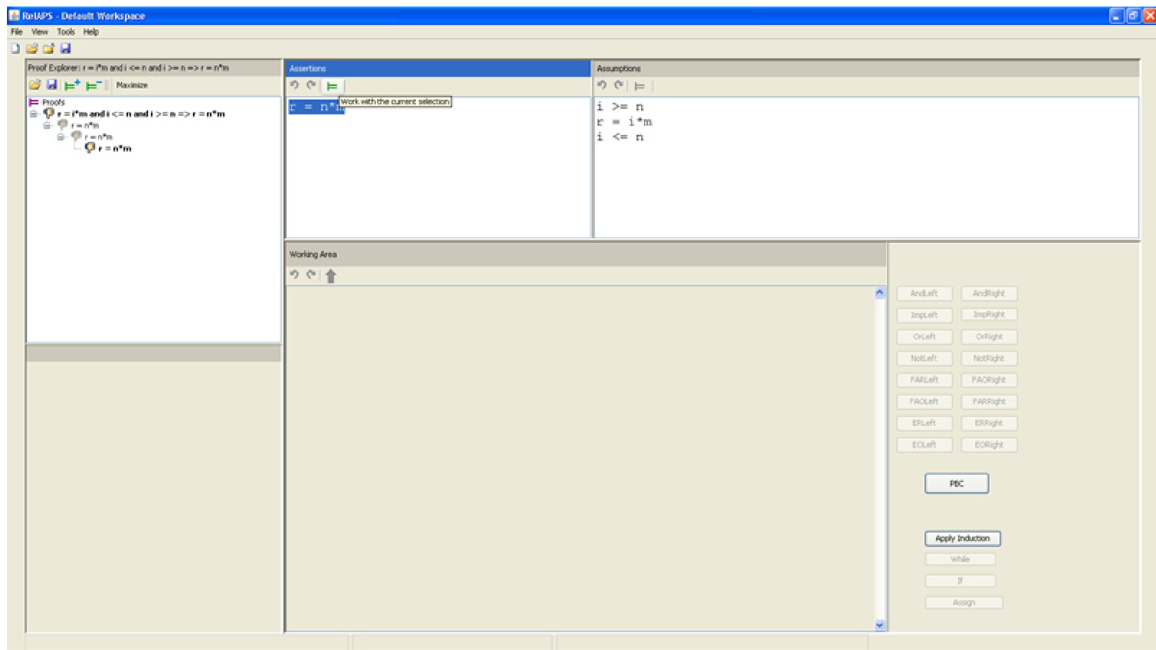

Figure 6.6: Selection of formula in `Assertions` window

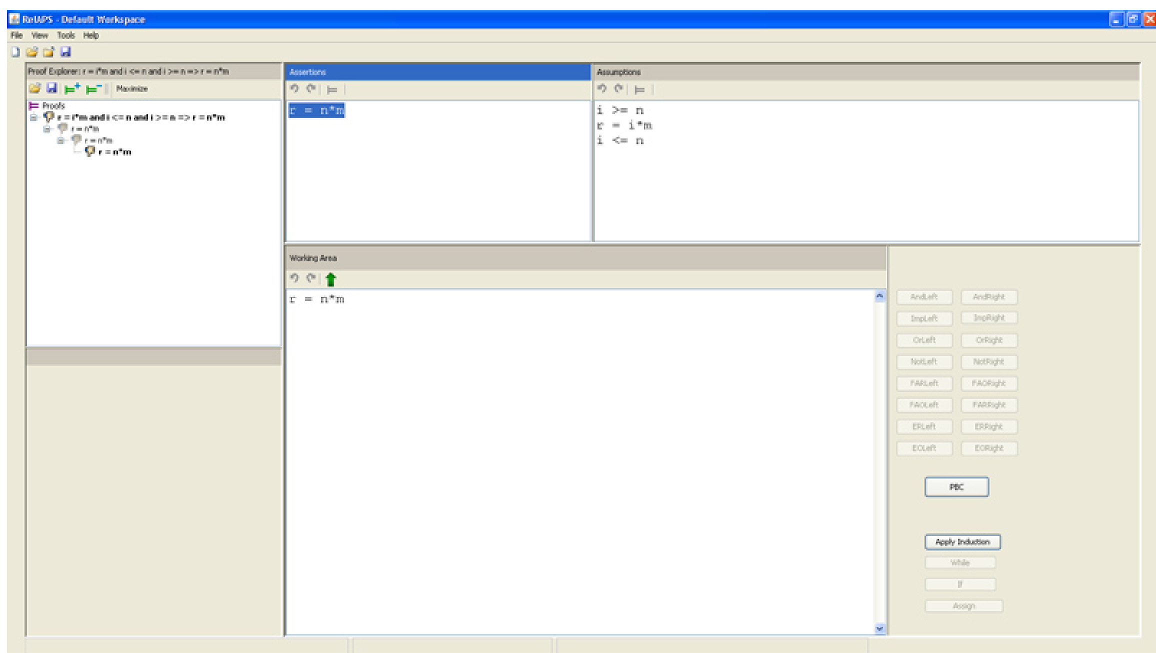

Figure 6.7: Application of derivation button in `Assertions` window

a <= b to the selected variable n of formula r=n*m, variable b in the left-hand side of the rule is automatically mapped to the selected variable n. So, to map the other variable, i.e., a, in the left-hand side of the rule, the user of this system will have to either select a term from the list, or enter a new term. Now, the selection of a term

from the list of available terms, i, for mapping of the variable a, will result in Figure 6.10.

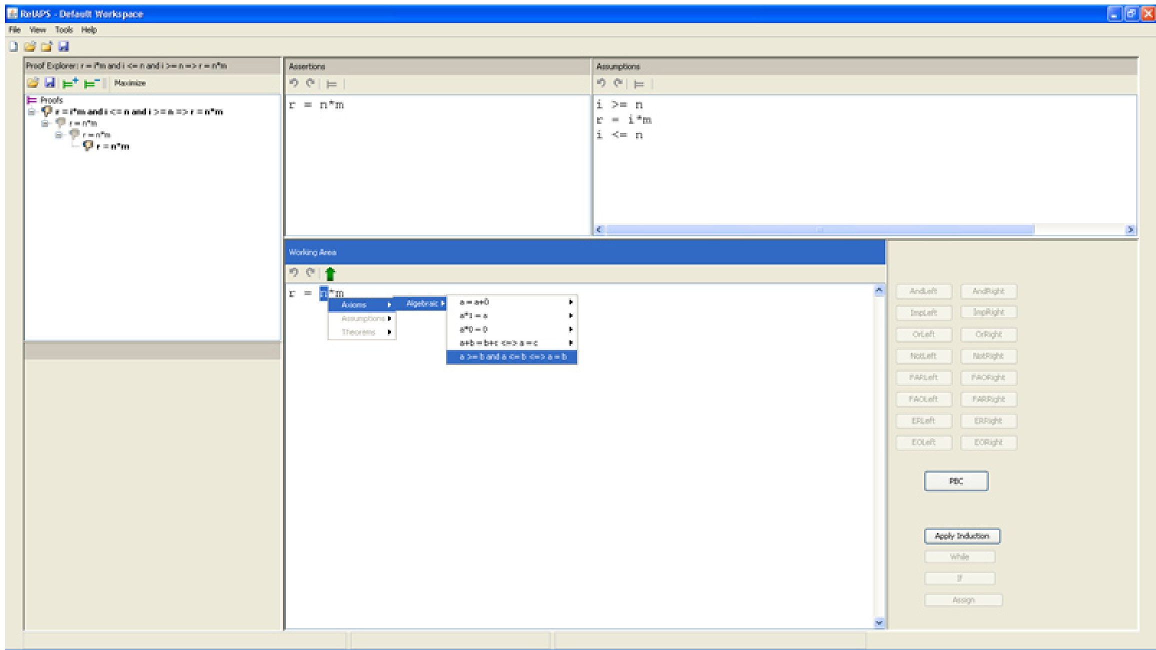

Figure 6.8: Application of an axiom rule

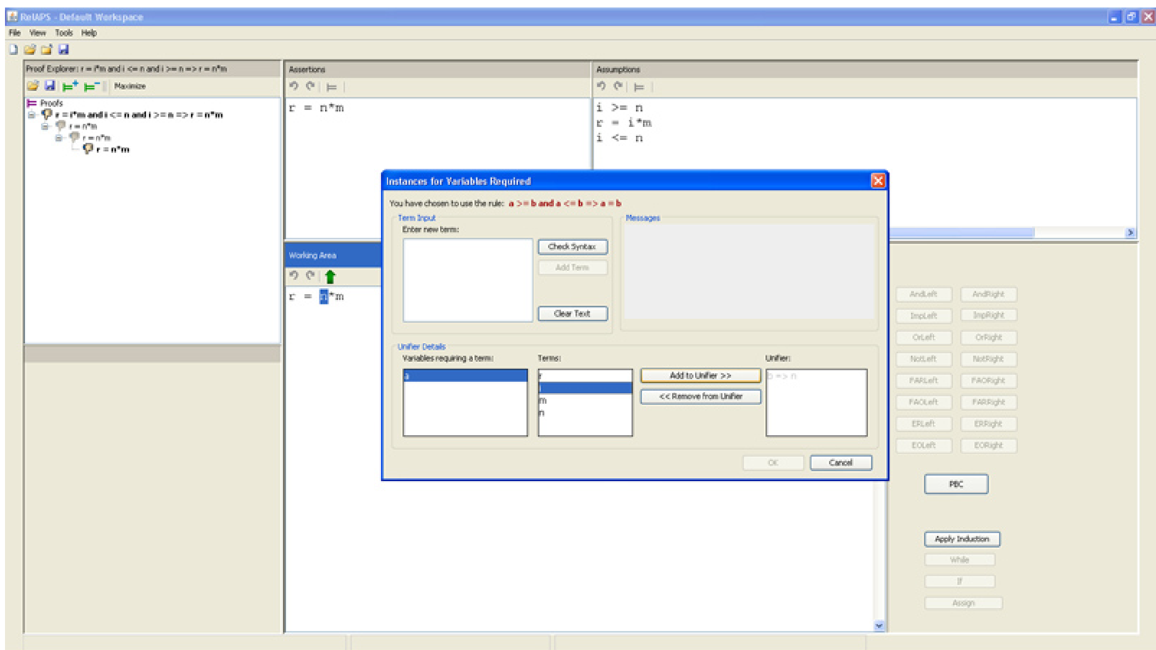

Figure 6.9: Mapping of variable a to i in unifier

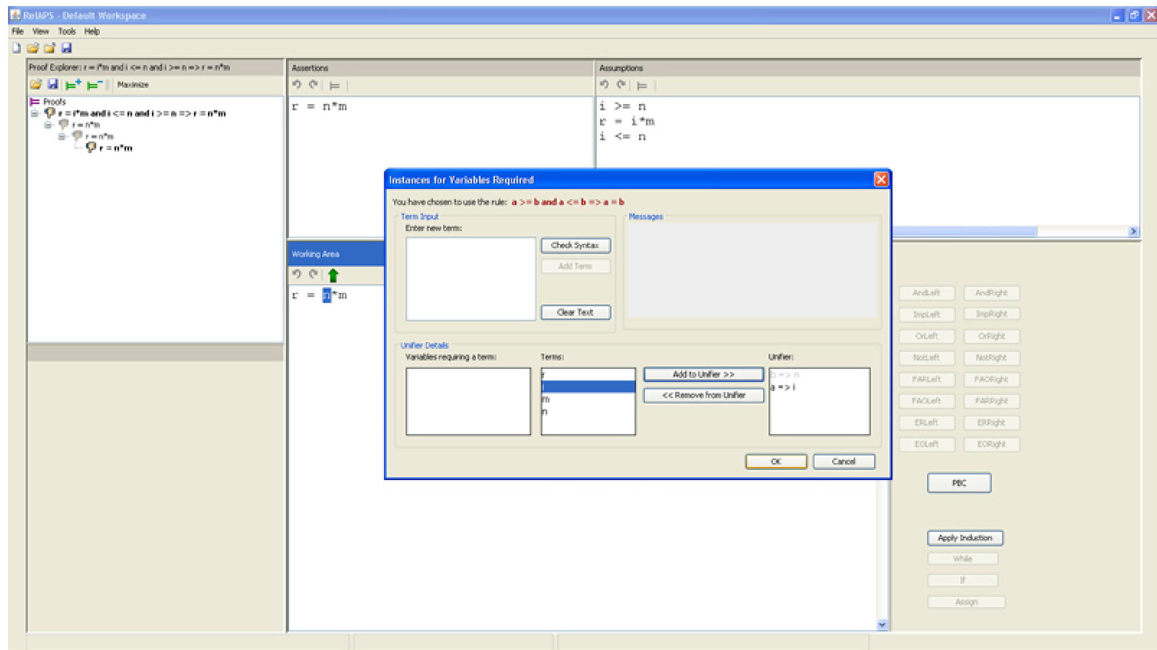

Figure 6.10: Mapping of variable a to i in unifier

The application of unifier is shown in Figure 6.11. As we can see, the atomic formula r=n*m has been converted to r=i*m, and for making it possible, two assumptions will be added to the current proof as we will see in Figure 6.12.

Now, if we click on derivation button of the `Working Area` window, we will have r=i*m in the `Assertions` window, and two new formulas, i >= 0 and i <= 0, will be added under the the main proof, r = n*m in the `Proof Explorer` window. As we have these two formulas, and the formula in the `Assertions` window as assumptions, the proof is done. This is demonstrated in Figure 6.12.

User of this system has the ability to maximize the `Proof Explorer` window to see a larger view of the generated proof tree. To use this feature, user will have to click on the `Maximize` button of the `Proof Explorer` Window. (Figure 6.13)

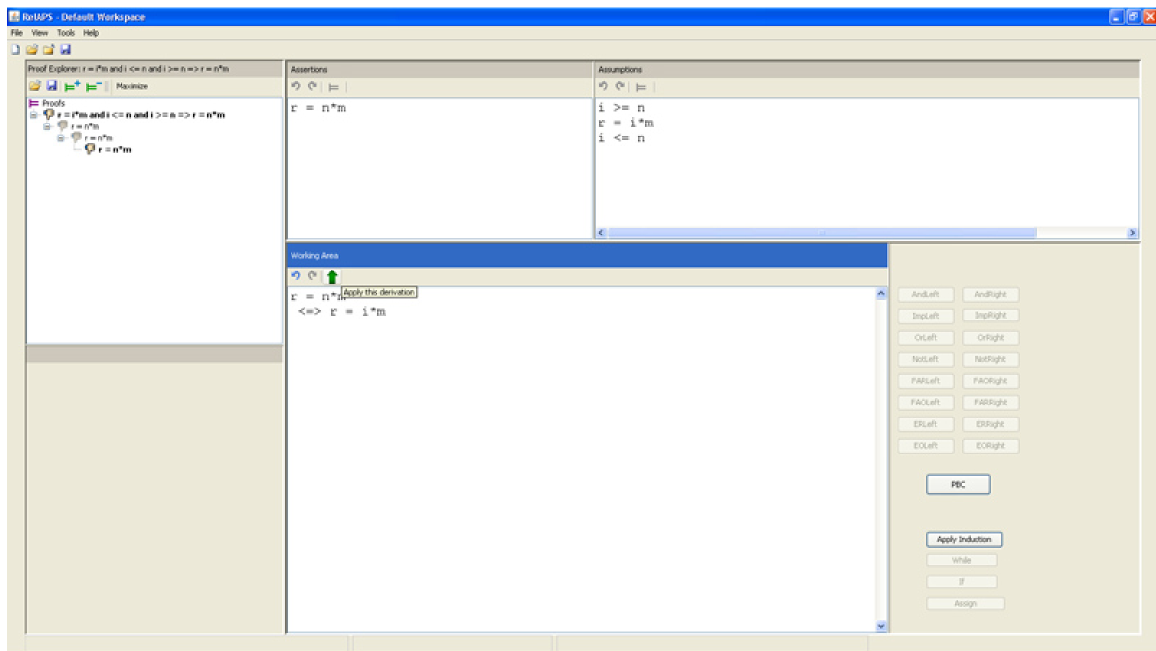

Figure 6.11: Application of unification

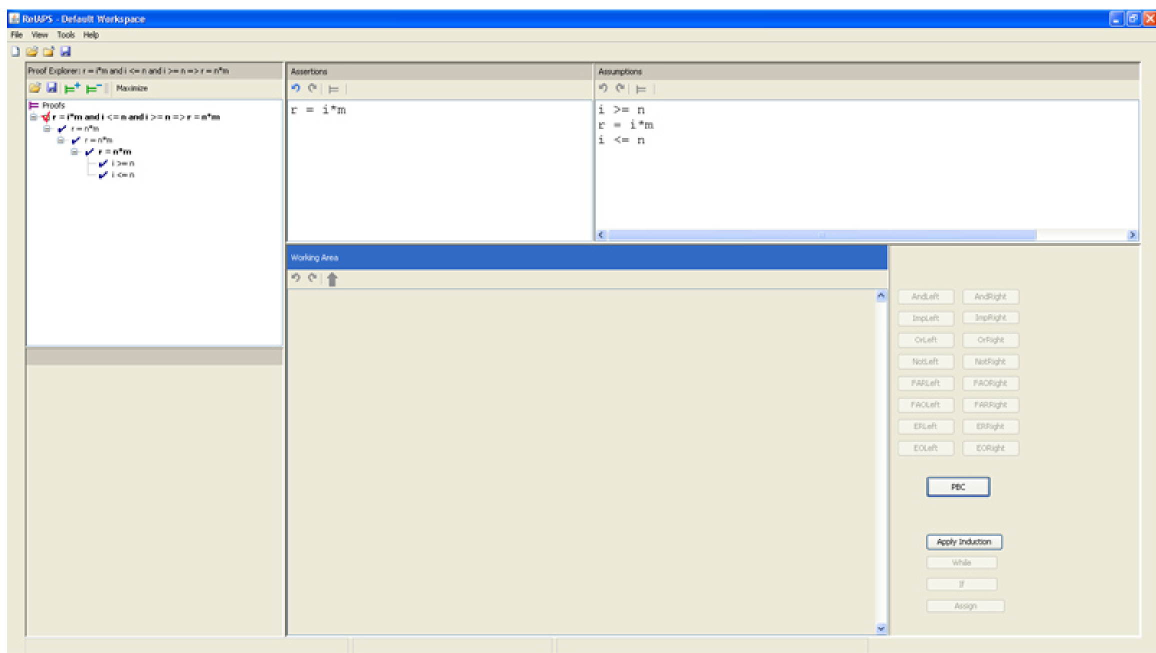

Figure 6.12: The completion of the main proof

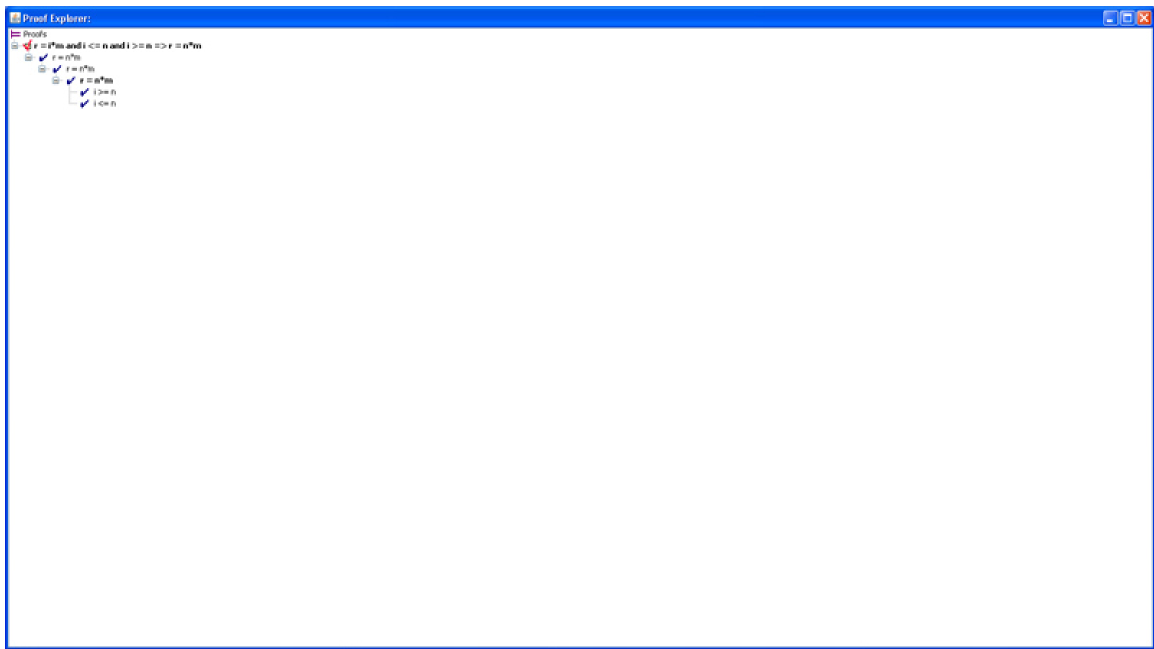

Figure 6.13: Maximized view of `Proof Explorer` window

Chapter 7

Conclusion

In this chapter we will briefly review the contents covered in this thesis and compare it with some other theorem proving systems. Then we will give some suggestions for extending the system in future.

7.1 Summarization of work

We started our discussion of basics of dynamic logic with the introduction of two modal operators of dynamic logic. Then we introduced the rules of Hoare logic and gave an example of program property verification using Hoare logic. Our discussion is followed by showing the relationship between Hoare logic and dynamic logic, where we presented how a dynamic logic formula can be used to represent the Hoare triple. Then our exploration of first order dynamic logic started with the introduction the syntactic materials of the language of our system. During the discussion of syntactic materials, we get familiarized with programs, formulas and test of first order dynamic logic. This is followed by giving semantics to the syntactic constructs using the arithmetic model we have used. In the end of that section, we discussed the satisfiability and the validity of formulas of dynamic logic. Then we compared two versions of reasoning in first-order dynamic logic: uninterpreted reasoning and interpreted reasoning. We discussed the interpreted version of reasoning that has been used in the system afterwards.

In Chapter 3, we have introduced the calculus of our system. In the first section, we presented the logical rules for first-order reasoning and inference rules for dynamic reasoning that we have implemented in this deductive system. This is followed by showing the soundness proof of these rules. In the end of this chapter, we have

talked a little bit about the incompleteness of this system.

In Chapter 4, we have discussed the strength of this system to be used for educational purposes. Here, we have shown the explicit visual explanation of some proof steps in this system.

Then, we explored the strategy we followed to implement this deductive system. We broke down our discussion of implementation in two parts. In the first part, we briefly explained our source code. Here we have learned about different packages of our system and how the system is responding to the user interations. This is followed by the discussion of class hierarchies where we presented the class diagrams of some important classes of our system. We also presented the relationship between some classes of our system.

Finally, we have presented a user manual for this system. Our intention was to make the system extremely user-friendly. For this reason, in that chapter, we showed how a proof can be carried out in this system with a detailed explanation. We presented every step of proving an example proof with the help of screenshots.

7.2 Comparison to other systems

The main purpose of developing this system is to meet the necessity of an educational tool for the beginners of program verification techniques. Examples of some of the existing systems which are very popular in this field of program verification is KIV tool and KeY system. These systems were mainly developed for commercial purposes. Different industries can use these system to fulfill the requirement of real program verification. For this reason, these systems are very much efficient in terms of the time and the space complexity, and the most of steps of program verification are automated there. So, these systems are not concerned about making everything easy and nice for the learners of the program verification techniques. On the other hand, in this system, we have chosen the calculli of the rules of the Hoare-style proofs which is very similar to how beginners learn. As we have used the weakest precondition transformation over the strongest postcondition transformation, we did not have to introduce any existentially quantified symbol for denoting the pre-assignment value of the changed variable, which is the case in KIV tool. Because of this feature, an assignment operation can be done merely by a substitution in the postcondition in

this system.

Some other features which make this system suitable to be used as an educational tool are the choice of the first-order calculus and the support of equational reasoning. As the first-order calculus we have chosen is based on natural deduction, it is very similar to doing it by hands. In addition to this, a user of this system has the flexibility to do equational reasoning exactly in the way we do it in a piece of paper. So, all these features make this system a suitable one over the existing program verification tools in terms of educational purposes.

7.3 Future work

There is a plenty of further work can be done to improve the capabilities of current system and to enhance it's robustness. The main focus in the future work should be on the inclusion of new data types and the automatic generation of candidates for loop invariants. Currenly in this system, only one data type is supported, namely intergers. More data types like character, double, float, array etc. can be added here to make it more robust. One major concern of developing a programming language verifier is loop invariant. In our system, we prompted the user of this system to enter a loop invariant. So, at present, it's the user's responsibility to provide the system with a valid loop invariant. But this feature can be updated by searching for some appropriate candidates for loop invariants and provide the user to select a particular loop invariant [9]. It will make the system more flexible and error free.

In this system, we have implemented only the rules for the partial correctness assertion. The rule for the total correctness assertion can be implemented in future. This feature will make this system complete if any complete model is used to interpret the syntactic materials.

Another possible work could be the addition of user-defined functions in this system. At present, this system does not support any procedure definitions in the program. So, there is a chance of improvement here by implementing this feature which will make it a true programming language. Only supported type of operation in the current system is INFIX. So, in future version, support for PREFIX and POSTFIX operations could be added.

Some other future improvements include the production of Latex output of the proofs, which will give the user the flexibility of exporting their work in the form of latex code. In that case, a researcher could use the system to prove the correctness of a program and could use the generated Latex output for publications.

Bibliography

[1] Aameri B., Winter M.: *A First-Order Calculus for Allegories*, In de Swart, H. (Eds.): Relational and Algebraic Methods in Computer Science (RAMiCS 12). Lecture Notes in Computer Science 6663, 74-91, (2011).

[2] Edsger W. Dijkstra: *Guarded commands, nondeterminacy and formal derivation of program*, Communications of the ACM, Volume 18 Issue 8, (August, 1975).

[3] Winfried Karl Grassmann, Jean-Paul Tremblay: *Logic and Discrete Mathematics*, Prentice Hall, ISBN-10: 0135012066, ISBN-13: 978-0135012062, Edition: 1, (Dec 28, 1995).

[4] R. Hahnle, M. Heisel, W. Reif, W.Stephan: *An interactive verification system based on dynamic logic*, In Proceedings of CADE, 306-315, (1986).

[5] David Harel, Dexter Kozen, Jerzy Tiuryn: *Dynamic Logic*, The MIT Press (2000).

[6] M. Heisel, W. Reif, W.Stephan: *Program verification using dynamic logic*, Lecture Notes in Computer Science, Springerlink, Volume 329/1988, 102-117 (1988).

[7] M. Huth, M. Ryan: *Logic in Computer Science*, Modelling and reasoning about systems, Cambridge University Press, 2nd Edition, (2004).

[8] Glanfield J., Winter M.: *RelAPS: A Proof System for Relational Categories*, In Schmidt, R.A., Struth, G. (Eds.): Relations and Kleene Algebra in Computer Science, 50-54, University of Sheffield CS-06-09, (2006).

[9] Laura Kovacs, Andrei Voronkov: *Finding Loop Invariants for Programs over Arrays Using a Theorem Prover*, FASE '09 Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, Pages 470 - 485 (2009).

[10] Benjamin Weib: *Deductive Verification of Object-Oriented Software*, KIV Scientific Publishing (2010).

[11] Wikipedia: "Modal Logic", Accessed 30th August, 2012.
http://en.wikipedia.org/wiki/Modal_logic

[12] Garson, James: "Modal Logic", The Stanford Encyclopedia of Philosophy (Winter 2009 Edition), Edward N. Zalta (ed.), Accessed 30th August, 2012.
http://plato.stanford.edu/entries/logic-modal

[13] Wikipedia: "Dynamic logic (modal logic)", Accessed 30th August, 2012.
http://en.wikipedia.org/wiki/Dynamic_logic_(modal_logic)

[14] Eric C.R. Hehner, University of Toronto: "Specifications, Programs, and Total Correctness", Accessed 30th August, 2012.
http://www.pm.inf.ethz.ch/education/seminars/se/presentations/Presentation_Eigenmann.pdf

[15] Wikipedia: "Hoare logic", Accessed 30th August, 2012.
http://en.wikipedia.org/wiki/Hoare_logic

[16] "Partial and total correctness", Accessed 30th August, 2012.
http://www.ida.liu.se/~TDDA43/slides.ax.sem/tot.corr.pdf

[17] Eric C.R. Hehner: "Specifications, Programs, and Total Correctness", Accessed 30th August, 2012.
http://www.cs.toronto.edu/~hehner/SPTC.pdf

[18] Wikipedia: "Loop invariant", Accessed 30th August, 2012.
http://en.wikipedia.org/wiki/Loop_invariant

[19] Gerwin Klein: "JFlex User's Manual", Accessed 30th August, 2012.
http://jflex.de/manual.html

[20] "UML basics: The class diagram", Accessed 30th August, 2012.
http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

[21] "The KeY Project", Accessed 30th August, 2012.
http://www.key-project.org/

[22] Wikipedia: "First order logic", Accessed 30th August, 2012.
http://en.wikipedia.org/wiki/First-order_logic

[23] "Completeness, incompleteness, and what it all means: first versus second order logic", Accessed 30th August, 2012.
http://lesswrong.com/lw/93q/completeness_incompleteness_and_what_it_all_means/

[24] "Collatz conjecture", Accessed 30th August.
http://en.wikipedia.org/wiki/Collatz_conjecture

[25] "ESC/Java2 Summary", Accessed 30th August.
http://kindsoftware.com/products/opensource/ESCJava2/

[26] "Boogie: An Intermediate Verification Language", Accessed 30th August.
http://research.microsoft.com/en-us/projects/boogie/