Verification of security protocols
tool support for update semantics

Nirmal, S.

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Verification of Security Protocols:

Tool Support for Update Semantics

by
Sajni Nirmal

Supervisors:

Dr. Erik de Vink (TUE)
Dr. Simona Orzan (TUE)

Eindhoven, August 2005

Abstract

Modelling security protocols has always been difficult and error prone. So, formal modelling becomes necessary. One proposed formal approach to model security protocols is Update Semantics. Update Semantics is a model-theoretic approach for reasoning about security protocols. But there is a practical drawback for this method. It is very complicated to use manually. This thesis offers a solution to this problem - namely tool support. So now it can be done automatically. The thesis explains in detail about the implementation and shows how easy the modelling of security protocols is with Update Semantics. The tool is flexible and easy to use. Moreover, it allows to visualize the epistemic states and also automatically verify the knowledge properties of security protocols. Modelling becoming easy with the update tool, it becomes more easy to visualize (with the help of existing established visualization tools) and also automatically verifying (with the help of classic model checking tools) the knowledge properties of the protocols. Three famous examples, the SRA Three Pass protocol, the Wide-Mouthed Frog protocol and the Needham Schroeder Key Exchange protocol are discussed in this thesis.

Contents

Chapter 1

Introduction

The role of security protocol is getting increasingly important in today's world. A protocol is a set of rules and conventions that define the communication framework between two or more agents. These agents, known as principals, can be end-users, processes or computing systems. A security protocol is a set of interaction rules aimed at guaranteeing some security property. These protocols are used to establish secure communication over insecure open networks and distributed systems. These protocols use cryptographic techniques to achieve goals such as confidentiality, authentication of principals and services, message integrity, non-repudiation, order and timeliness of the messages, and distribution of cryptographic keys. Unfortunately, open networks and distributed systems are vulnerable to hostile intruders who may try to subvert the protocol design goals. The basic notion of these protocols are that participants exchange messages, and always assume a hostile communication environment represented by an intruder.

Designing these security protocols is error prone and difficult. Proving the correctness of these protocols have so far been of limited success. It has been recognized that formal methods and automatic analysis play an important task in the modern design of security protocols, since doing it manually is very much erroneous. Several methods are being used to prove the correctness like the so-called BAN Logic [BAN96]. Many useful results have been reported, but due to their complexity, it has very limited success.

One of the proposed approach is Update Semantics [Hom03, HMV05]. It is a model-theoretic approach for reasoning about security protocols, applying recent insights from dynamic epistemic logics. It focuses on describing exactly the subsequent epistemic states of the agents participating in the protocol, using Kripke models and transitions between these based on updates of the agents' beliefs associated with steps in the protocol. The main intention is not to prove that the protocol is completely secure, but if the participating agents are honest, then the intruder does not learn anything from a single run of that protocol.

Moreover, it can show how much an intruder can learn about the agents participating. Update Semantics looks very promising in analyzing the security protocols from the point of view of information exchanged. But then again, there is a practical drawback. The number of agents and/or the quantity of information present in the protocol can increase and if it does, modelling it manually is not a good solution.

This thesis offers a solution for this problem, namely a tool support for applying the rules in Update Semantics. An update tool has been implemented to make modelling easier. The Kripke models used in Update Semantics are encoded as Labelled transition systems (LTSs), and the update rules are implemented as mechanical transformations on these encodings. The update tool is very much user friendly. Another feature of this tool is that the output generated from this tool is very much compatible with several existing visualization and verification tools.

In short words, given the Kripke model, it is possible to find out what is the belief of each agent participating in the security protocol within a very short time. It is very difficult to generate each stage of the model by hand and then convert them manually into LTS, and then verify them or visualize them. It is very easy to make mistakes. The examples given in the paper on Update Semantics [HMV05] are drawn manually and not with the help of any tool. Slight mistakes were found out in some of the pictures that was drawn in that paper. The correct version of those pictures were generated using the update tool.

1.1 Overview

This thesis is divided into 5 chapters. In Chapter 2, definitions of Kripke models and how they are encoded to LTS (labelled transition system) are given. Chapter 3 covers all the transformations in Update Semantics. Detailed explanation of the rules are provided in that chapter. Chapter 4 explains the implementation of the update tool. A detailed description of how each rule is implemented is explained. In Chapter 5, the visualization and verification of the protocols is explained. It is shown by means of some examples how the tool enables the visualization and verification of security protocols. Finally in Chapter 6, the conclusions are given.

Chapter 2

Preliminaries

This thesis is mainly based on the paper "Update Semantics Of Security Protocols" [HMV05], where the Update Semantics theory is introduced. So, the first step was to read and understand the paper. It is a model-theoretic approach for reasoning about security protocols and *Kripke models* are used to model it.The key idea is to encode the participants' beliefs in a Kripke model and use update steps between models to show how participants update their beliefs at every step of the specified protocol.

The first section gives the definition and shows how the knowledge is represented using Kripke models. The second section gives an explanation of Labelled Transition System (LTSs). The last section explains how Kripke models are encoded in the LTS format.

2.1 Knowledge Representation using Kripke Models

The definition of Kripke model is given here.

Definition 2.1.1 *Let AP be a set of propositions. A Kripke model (or structure) is a tuple* $(\langle S, \pi, R_1, \ldots, R_m \rangle, s)$*, where:*

- S *is a set of states* {possible worlds}
- $\pi : S \times AP \rightarrow \{\texttt{true}, \texttt{false}\}$ a function saying which propositions are true in each state {valuations}
- $R_1, \ldots, R_m : S \times S$ are relations {agents}
- $s \in S$ is the initial state {real world}

The Kripke structure represents an *epistemic state* of a system of agents. S contains all the states that are being modelled. The state represents a possible

world for an agent. An epistemic state represents all the possible worlds for the agents involved. The initial state shows the real world and all the other states are possible worlds for the agents involved. The pointed arrow shows the real world (starting state). π (valuation)defines the situation in every world. It defines what the agents sees/ believes in each state. The propositions represent keys, secrets, messages etc. The relations shows indistinguishability. Those are the relations for agents. There is a relation from a state(world) to another state(world) for an agent a only if it considers that world possible. There is one initial state that shows the real world and it is shown with the pointed arrow, see Example 2.1.1.

For analyzing security protocols, it is assumed that we have total knowledge of the values of the variables in different runs of a protocol. For example, the program variable p in a protocol run has the value $[\![p]\!]$. In the real world it is, obviously, always true that $p = [\![p]\!]$. We will abbreviate $p = [\![p]\!]$ to p on (thus transforming a program expression into a propositional variable). Similarly, $\neg p$ is an abbreviation of $p \neq [\![p]\!]$. For example, agent a that learns $B_b p \vee B_b \neg p$, learns that agent b has assigned a value to the program variable p.

Example 2.1.1 *An example of Kripke model,* $(M, s) = (\langle S, \pi, R_1, \ldots, R_m \rangle, s)$ *is shown in Figure 2.1.*

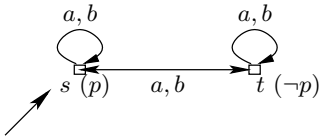

Figure 2.1: $(M, s) = (\langle \{s, t\}, \pi, a, b, \rangle, s); \pi(s)(p) = \texttt{true}$, $\pi(t)(p) = \texttt{false}$

In this model, a and b are the agents. Normally, a, b, c, etc. are used as typical agents, taken from a class $\mathcal{A}$ (set of all agents). s shows the real world and t shows the possible world for these agents. The valuation for each state is given as, $\pi(s)(p) = \texttt{true}$, $\pi(t)(p) = \texttt{false}$. The set of propositions is the singleton set $\{p\}$. This leads to possible worlds, one where p is true (s), the second where p is false (t). This Kripke structure models the fact that p is true in reality by having s marked as initial state. Moreover, none of the agents knows what the reality is. This is marked by the arrows a, b. If agent a, for instance, is placed in the world s, he believes that both worlds s and t are possible, meaning he does not leave enough information to distinguish which one is the real world. Evaluating the beliefs of agents on a Kripke model is explained formally below. The set of propositional variables in a model is denoted as $\mathcal{P}$.

Definition 2.1.2 *The class of objective formulas is the smallest class such that*

- *all propositional variables and atoms* $p \in \mathcal{P}$ *are objective;*

- *if ϕ is objective, then $\neg\phi$ is objective;*
- *if ϕ_1 and ϕ_2 are objective, then $\phi_1 \wedge \phi_2$ is objective.*

Belief formulas: The valuations for the states are objective formulas. B is used as a belief modal operator. For example, $B_a\phi$ should be read as 'a believes ϕ'. This is interpreted as formulas on standard Kripke models $(M, s) = (\langle S, \pi, R_1, ..., R_m\rangle, s)$, where $(M, s) \models B_i\phi$ iff $\forall t \in S\colon R_i(s, t) \rightarrow (M, t) \models \phi$. This means for a Kripke model (M, s), in a state s, an agent i believes ϕ if and only if all the relations from that state reaches a state where the valuation is ϕ. Also, In any world, we cannot have both $B_i\phi$ and $\neg\phi$, for an objective formula ϕ.

2.2 Model Checking

The use of Kripke structures as presented above is not the only existent use. Kripke structures are also widely used in a rather different context: formal verification of critical systems. There, Kripke structures represent the possible behaviors of such a system: states are simple states, the valuation describes the states; there is only one relation, the transition relation, which models the evolution of the system in time; s and t are related if the system can evolve in one move from state s into state t.

Another representation for the behavior of evolving agents, focussing on the transitions instead of states, are LTSs. The definition of LTS (Labelled Transition System) is given below.

Definition 2.2.1 *Let Act be a set of labels. A labeled transition system (LTS) is a triple (S, T, s), where:*

- *S is a set of* states
- *$T \subseteq S \times Act \times S$ a set of labeled* transitions
- *$s \in S$ is the* initial state

In an LTS, S is the set of all states. Each transition in LTS from any one state to any other state is labelled. There is always an initial state.

The Kripke models used by the Update Semantics are multi-relational, i.e. there is a relation for every participating agent. In other words, we have labels both on states and transitions. As we have seen, most existing model checkers operate on either one-relational Kripke structures (labels on states, no labels on transitions) as shown in Figure 2.2 or LTSs (labels on transitions, no labels on states) as shown in Figure 2.3. Therefore, some encoding is necessary in order to obtain a multi-relational Kripke model as shown in Figure 2.4.

We choose to encode the multi-relational Kripke structures as LTSs and thus get access to the extensive tool support (visualization, verification) available for LTSs. The transformation tools contributed by this thesis take LTS as input and produces LTS as output.

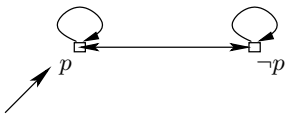

Figure 2.2: One-relational Kripke Structure

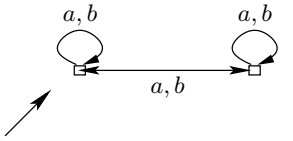

Figure 2.3: Labelled Transition System (LTS)

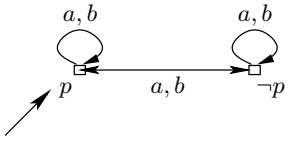

Figure 2.4: Multi-relational Kripke structure

2.3 Encoding the Kripke models into LTS format

The states in the Kripke model will be changed to natural numbers while encoding in the LTS format. There are two types of transitions: agent transitions labelled a,b and knowledge transitions labelled K_p. The latter type will represent the π function from the original Kripke model. The valuation function such as $\pi(s)(p) = \texttt{true}$ in the Kripke model will be encoded as (s,K_p,s). 0 is always the starting state (root) and will represent the real world.

An example is shown below in Figure 2.5 to get a better understanding of how a *Kripke model* is encoded into a LTS. The states in the Kripke model (s,t) are now 0 and 1. After converting to LTS, they are stored in an AUT format. An AUT file describes an LTS in text format. Each state is represented by a natural number.The first line of an AUT file looks like this:

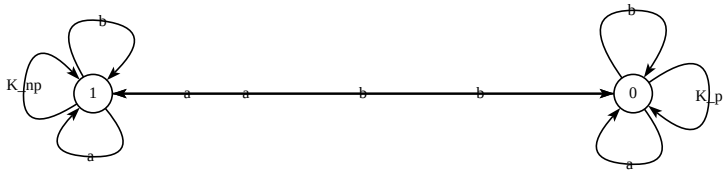

Figure 2.5: Kripke model encoded in LTS format

```
des (0, <number-of-transitions>, <number-of-states>)
```

Each of the lines that follow represents a transition and looks like this:

```
(<initial state>, <label>, <final state>)
```

where `<initial state>` and `<final state>` are numbers and `<label>` is a case-sensitive character string (up to 5000 characters). 0 is always the initial state.

```
des(0,10,2)
(0,a,0)
(0,b,0)
(0,a,1)
(0,b,1)
(0,K_p,0)
(1,a,1)
(1,b,1)
(1,a,0)
(1,b,0)
(1,K_np,1)
```

`The AUT format of the LTS shown in the figure above.`

An AUT format is used for several reasons. One of the reasons is that many visualization tools accept this AUT format as input. For this project some visualization tools were tried, like dotty from GraphViz [Gra04], and also CADP [FGK+96]. So, as soon as we generate each stages of the protocol (the output generated using the code is also in LTS stored in AUT format), they can be visualized in 2D format. Several tools for visualization accept AUT format for input. Another reason to use this format is that automatic verification tools like CADP model checker also accepts it. That means, as each stage is being generated, the belief of each agent can be automatically verified.

Chapter 3

Update Semantics

We have seen in (Section 2.1) how an epistemic state is represented as a Kripke model. We now need a mechanism to express transitions from one epistemic state to another as a consequence of actions happening in the system. (In the case of a security protocol, the actions are exchanging messages.) A transition from one epistemic state to a new one after learning some fresh piece of information is called a *belief update*. There are three types of belief updates that we consider (following [HMV05]):

- public announcement of a propositional variable,
- private learning of a variable, and
- private learning about the knowledge of other agents.

The first type of update typically runs as follows: In an open network, agent a sends a message to agent b. It is customary to assume that all agents in the network can read this message too. In contrast, the second type of update, describes private learning. For example, agent b receives a message $\{x\}_k$ from agent a. (Here, we use the notation $\{x\}_k$ to denote a message x encrypted with the cryptographic key k.) If b possesses the key k, then b privately learns the message content x. The final type of update is probably the most interesting. It is realistic to assume that the steps in a protocol run are known to all agents. Therefore, observing that an agent receives a message will increase the knowledge of the other agents. For example, if agent a sends a message $\{x\}_k$ to agent b, then agent c learns that b has learned the information contained in the message $\{x\}_k$, but typically, c does not learn x if c does not possess the key k.

In Section 3.1, we introduce a series of mechanical transformations that need to be done on Kripke models in order to achieve the belief updates mentioned above. After that, in Section 3.2 we explain how the transformations can be employed to model runs of security protocols.

3.1 Operations on Kripke models

The belief updates considered can be made concrete by defining some convenient operations on Kripke models. The main operations are UPDATE (Subsection 3.1.1) and SIDE-EFFECT (Subsection 3.1.2). In short, UPDATE realizes private learning of a formula and SIDE-EFFECT, private learning about the knowledge of other agents. Public learning can be modelled using UPDATE. A third operation, ADDATOM, allows the enlargement of the universe of known variables. The presentation in this section is based on [HMV05] and we refer to that paper for further details.

The definitions of these operations are rather complex. Because the goal in this thesis is to arrive at a correct implementation of them, we first extract a transformation algorithm from every definition. In the next chapter, these algorithms and their implementations will be discussed in detail.

3.1.1 UPDATE

Update is the private learning of a formula. The public learning of a message can also be modelled using update. Whenever a message is sent as a public announcement through the network, all the agents concerned learn about it and update their knowledge beliefs. Whenever an encrypted message is sent for a specific agent, that agent learns the content of the message privately. Both these situations are taken care of with this update function.

Definition 3.1.1 *Let a model* $(M, w) = (\langle S, \pi, R_1, ..., R_m \rangle, w)$*, a group of agents* $\mathcal{B}$*, and an objective formula* ϕ *be given. Then the model* $\text{UPDATE}_{(\phi,\mathcal{B})}(M, w)$*, the update of* (M, w) *for agents in* $\mathcal{B}$ *and formula* ϕ*, is given by*

$$\text{UPDATE}_{(\phi,\mathcal{B})}(M, w) = (\langle S', \pi', R'_1, ..., R'_m \rangle, w')$$

where

- $S' = \{\ old(s),\ new(s) \mid s \in S\ \}$
- $w' = new(w)$
- *for all* $p \in \mathcal{P}$*:* $\pi'(old(u))(p) = \pi'(new(u))(p) = \pi(u)(p)$
- *for* $a \in \mathcal{A}$*, the binary relation* R'_a *on* S' *is minimal such that*

$$\begin{array}{lcll} R'_a(old(u), old(v)) & \Leftrightarrow & R_a(u, v) & \\ R'_a(new(u), new(v)) & \Leftrightarrow & R_a(u, v) \wedge (M, v) \models \phi & \text{if } a \in \mathcal{B} \\ R'_a(new(u), old(v)) & \Leftrightarrow & R_a(u, v) & \text{if } a \notin \mathcal{B} \end{array}$$

Algorithmic Explanation of the Definition

S' is the set of states in the new model. It contains the states in the *old* model and also the newly created states. It also includes the new starting point. w'

is the new starting point.For the update rule, the old model is taken and one duplicate copy of it is made. So, the output model contains the old states and the transitions along with its new copy. So, the old states and transitions are tagged with *old* and the duplicates are tagged *new* to differentiate between them. The states in the old model are duplicated to make the new model. So, the valuations of the states from the old model and its new duplicates will have the same valuation.

To obtain the relations in the new model these three rules are applied.

1. For all agents, $a \in \mathcal{A}$, if there was a relation from any one of the old state to any other old state, then they are included in the output model.
2. For all agents, $a \in \mathcal{A}$, and $a \in \mathcal{B}$, if there was a relation from state u to state v in the input model, and valuation of state v is ϕ, then there is a relation from $new(u)$ to $new(v)$ in the output model.
3. For all agents, $a \in \mathcal{A}$, and $a \notin \mathcal{B}$, if there was a relation from state u to state v in the input model, then a relation exists from $new(u)$ to $old(v)$ in the output model.

3.1.2 SIDE-EFFECT

SIDE-EFFECT is the private learning about the knowledge of other agents. This happens because it is assumed that all agents in the network can read messages that are being sent and received. So, if agent a sends a message $\{x\}_k$ to agent b, then all the other agents involved learns that agent b has learned the information contained in message $\{x\}_k$.

To learn about the side-effect of agents, several subtransformations have to be performed. First step in this process is UNFOLD. States are shared among agents. When a state is changed, in order to change the belief of one agent, the beliefs of other agents are affected by that. So, first the agents who learn are separated from agents that don't learn. Then the next step in the process is finding the partial submodel, (SUB). Partial submodel represents the knowledge of the learning agents. This is mainly done to make the third step easier, namely ATOMSPLIT. This subtransformation is done only for agent who learn. So, ATOMSPLIT is only applied to the partial submodel. When all these steps are performed, the side-effect can be learnt.

UNFOLD

Unfolding is the process of separating the states of learning agents from the states of agents that do not learn. As states are shared among agents, it is obvious that if a state is changed with the intention to change the belief of one agent, then the belief of the other agents that consider this state possible is

changed as well. Therefore, the first thing to do, is to separate the states of learning agents from the states of agents that do not learn. This procedure is called unfolding.

Definition 3.1.2 *Given a model* (M, w) *with* $M = \langle S, \pi, R_1, ..., R_m \rangle$*, and a partitioning* $\mathcal{X}$ *of* $\mathcal{A}$*, we define the operation* $\text{UNFOLD}_{\mathcal{X}}(M, w)$*, the unfolding of* (M, w) *with respect to* $\mathcal{X}$*, by* $\text{UNFOLD}_{\mathcal{X}}(M, w) = (\langle S', \pi', R'_1, ..., R'_m \rangle, w')$*, where*

- $S' = \{\ new_{\mathcal{B}}(s) \mid s \in S,\ \mathcal{B} \in \mathcal{X}\ \} \cup \{orig(w)\}$
- $w' = orig(w)$
- $\pi'(new_{\mathcal{B}}(s))(p) = \pi(s)(p)$ *and* $\pi'(orig(w))(p) = \pi(w)(p)$ *for all* $s \in S$*,* $p \in \mathcal{P}, \mathcal{B} \in \mathcal{X}$
- *for* $a \in \mathcal{A}$*, the binary relation* R'_a *on* S' *is minimal such that*

$$\begin{array}{lcl} R'_a(new_{\mathcal{B}}(s), new_{\mathcal{B}}(t)) & \Leftrightarrow & R_a(s,t) \\ R'_a(orig(w), new_{\mathcal{B}}(s)) & \Leftrightarrow & R_a(w,s) \text{ and } a \in \mathcal{B} \end{array}$$

where $\mathcal{B}$ *ranges over* $\mathcal{X}$*.*

Algorithmic Explanation of the Definition: S' is the set of states in the new model. It contains the states in the original model and also the newly created states. It also includes the new starting point. w' is the new starting point. The states and transitions in the old model are duplicated for each $\mathcal{B} \in \mathcal{X}$. The valuation remains the same for all the states and its duplicates in the new model. Also the new starting point has the same valuation as the old starting point. Also for every $\mathcal{B} \in \mathcal{X}$ and every $a \in \mathcal{B}$, if there was an arrow(transition) labelled a from $w \rightarrow s$ in the old model, then an arrow (labelled a) is added from the new starting point to the copy of s belonging to the duplicated model corresponding to $\mathcal{B}$.

For applying the side-effect for a specific set of agents, the idea is to separate their knowledge from the rest of the agents without compromising anything. $\mathcal{A}$ is the set of agents and $\mathcal{B}$ is the group of agents whose side-effect should be learnt. Using UNFOLD, the knowledge of $\mathcal{B}$ is separated from the rest. Next step is finding the partial submodel, ($\text{SUB}_{\mathcal{B}}(M)$)that represents the knowledge of the group of agents in $\mathcal{B}$.

Partial submodel ($\text{SUB}_{\mathcal{B}}(M)$)

Given a model that is obtained from unfolding the original model, the definition explains how to find the sub model of $\mathcal{B}$. This partial submodel represents the knowledge of agents in $\mathcal{B}$. This is mainly for handling the ATOMSPLIT operation, that is explained later on in this chapter.

Definition 3.1.3 *Given a model* (M, w) *such that*

$$(M, w) = (\langle S, \pi, R_1, ..., R_m \rangle, w) = \text{UNFOLD}_{\mathcal{B}, \mathcal{A} \setminus \mathcal{B}}(M', w')$$

for some (M', w')*, define* $\text{SUB}_{\mathcal{B}}(M)$*, the submodel of* M *for* $\mathcal{B}$*, by* $\text{SUB}_{\mathcal{B}}(M) = \langle S', \pi', R'_1, ..., R'_m \rangle$ *where*

- $S' = \{\ new_{\mathcal{B}}(s) \mid s \in S\ \} \cup \{orig(w)\}$
- $\pi'(s)(p) \Leftrightarrow \pi(s)(p)$ *for all* $s \in S'$, $p \in \mathcal{P}$,
- *for all* $a \in \mathcal{A}$, $R'_a(s,t) \Leftrightarrow R_a(s,t) \wedge s, t \in S'$.

Algorithmic Explanation of the Definition: S' is the set of states showing the knowledge of agents in $\mathcal{B}$ (the specific partition for the agents in $\mathcal{B}$ after unfolding) and also the starting state. The valuation of all the states remains the same. For all agents $a \in \mathcal{A}$, a relation from $s \rightarrow t$ exists in the partial submodel, only if it existed in the model obtained after applying UNFOLD and also both s and t should be in S.

By finding out the partial submodel, it actually isolates the knowledge of agents in $\mathcal{B}$. Once those states are isolated, it is much easier to find out the side-effect. the partial submodel highlights the knowledge of agents in $\mathcal{B}$. There is a restmodel(REST_N) which covers the rest of the model. It is explained in the Update Semantics paper [HMV05] in more detail.

ATOMSPLIT

Next step in the process is ATOMSPLIT. It is applied only to the partial submodel. The operation $\text{ATOMSPLIT}_{(\phi, \mathcal{B})}$ removes the arrows for agents in $\mathcal{B}$ between the states that have different valuation for the objective formula ϕ.

Definition 3.1.4 *Given a model* $M = \langle S, \pi, R_1, ..., R_m \rangle$*, an objective formula* ϕ *and a set of agents* $\mathcal{B}$*, we define the operation* ATOMSPLIT *by*

$$\text{ATOMSPLIT}_{(\phi, \mathcal{B})}(M) = \langle S', \pi', R'_1, ..., R'_m \rangle$$

where

- $S = S'$,
- $\pi'(s)(p) \Leftrightarrow \pi(s)(p)$ *for all* $s \in S'$, $p \in \mathcal{P}$,
- *for* $a \in \mathcal{B}$, $R'_a(s,t) \Leftrightarrow R_a(s,t) \wedge M, s \models \phi \Leftrightarrow M, t \models \phi$,
- *for* $a \notin \mathcal{B}$, $R'_a(s,t) \Leftrightarrow R_a(s,t)$.

Algorithmic Explanation of the Definition: The set of states S remains the same. The valuation of the states also remains the same. For every agent $a \in \mathcal{A}$, a relation from $s \to t$ exists if it existed in the input model and both s and t should have the same valuation for ϕ. For all agents, $a \notin \mathcal{B}$, an arrow from $s \to t$ labelled a exists in the new model if it existed in the input model.

Finally, the definition of side-effect function.

Definition 3.1.5 *For a model (M', w'), a set of agents $\mathcal{B}$ and an objective formula ϕ such that $(M', w') = \text{UNFOLD}_{\{\mathcal{B}, \mathcal{A}\setminus\mathcal{B}\}}(M, w)$ and $N = \text{SUB}_{\mathcal{B}}(M')$ we define the operation $\text{SIDE-EFFECT}_{(\phi,\mathcal{B},\mathcal{C})}(M, w)$, the side-effect for agents in $\mathcal{B}$ with respect to the agents in $\mathcal{C}$ and the formula ϕ, by*

$$\text{SIDE-EFFECT}_{(\phi,\mathcal{B},\mathcal{C})}(M, w) = (\text{REPLACE}_N(\text{ATOMSPLIT}_{(\phi,\mathcal{C})}(N), M'), w').$$

3.1.3 ADDATOM

Adding a fresh propositional variable to the model is accomplished by this operation. This is done by making two copies of the original states. One of them will be assigned "positive" and the other as "negative". In the positive states, proposition will be true and false in the negative states.

Definition 3.1.6 *Given a model $(M, w) = \langle S, \pi, R_1, ..., R_m\rangle$ and a fresh proposition p, we define the operation ADDATOM_p such that $(M', w') = \text{ADDATOM}_p(M, w) = \langle S', \pi', R'_1, ..., R'_m\rangle$ where*

- $S' = pos(S) \cup neg(S)$
- $\pi'(pos(s))(q) =$ *if* $p = q$ *then* `true` *else* $\pi(s)(q)$
- $\pi'(neg(s))(q) =$ *if* $p = q$ *then* `false` *else* $\pi(s)(q)$
- $R'_i(s, t) \Leftrightarrow R_i(s, t)$ *for any agent* i
- $w' = pos(w)$

Algorithmic Explanation of the Definition

S' is the set of all states in the positive and negative copies of the original states. The valuation of the new proposition p is true in the positive copy of the state and the valuations for all other propositions remain the same for that particular state. The valuation of the new proposition p is false in the negative copy of the state and the valuations for all other propositions remain the same for that particular state. Whatever relations existed for a particular agent in the input model remains the same after this operation. w' is the new starting point, i.e the positive copy of the old starting point.

3.2 Modelling security protocols in Update Semantics

Security protocols are series of messages exchanged between a set of agents, in order to achieve some security goal like secret communication or mutual authentication. When analyzing security protocols, it is always assumed that a malicious entity exists, with the power to intercept, delete and create messages (the *intruder*). The analysis of security protocols based on update semantics allows to see, at every stage of the protocol, the exact information that each participating agent and the intruder have learned until that stage. In this way, it is visible for instance whether a secret piece of information has leaked to the intruder.

The goal of such an analysis is not to prove that a given protocol is completely secure, but rather to show exactly what information the intruder was able to collect. For this analysis, there should be a proper methodology of how and when to apply the rules that are mentioned in the section above. We propose a modelling methodology. It is shown in the table below.

Protocol	Update Semantics
s has knowledge m	ADDATOM_m $\text{UPDATE}_{(m,s)}$
$a \to b : m$	$\text{UPDATE}_{(m,*)}$
$a \to b : \{m\}pkb$ (a has knowledge $\{m\}pkb$; public learning of variable)	$\text{UPDATE}_{(\{m\}pkb,*)}$ $\text{UPDATE}_{(m,b)}$ $\text{SIDE-EFFECT}_{(m,*,b)}$
$a \to b : \{m\}pkb$ (a has knowledge $\{m\}pkb$; private learning of variable)	$\text{UPDATE}_{(\{m\}pkb,*)}$ $\text{UPDATE}_{(m,b)}$

According to the first column, whenever a situation arises to show that a particular agent has a specific knowledge, it is modelled using two steps. First the knowledge is added to the model, and then updated for that specific agent. The second column shows that whenever an unencrypted message is sent through the network, it can be seen by all, that means, whoever sees it gets their knowledge updated. This is modelled with one update step. The third column shows the public learning of propositional variables. Whenever an encrypted message is sent through the network, it can be seen by all, that means, whoever sees it gets their knowledge updated with the fact that a message is being sent. And, the specific agent who has the key learns the content of the message. So, then everybody else in the network knows that the specific agent has learnt the content. This is modelled with three steps, one update step to show that everyone learnt the encrypted message, then another update step to the private learning of the content, and one side-effect step to know the side-effect of the private learning of other agents. Private learning of the variable is shown next.

Chapter 4

Implementation Methodology

This chapter discusses about the implementation of the update tool. It also describes the important decisions taken during implementation. One global function is made from which calls can be made to all the functions that are implemented. It contains a help file, which shows how the input commands should be given and also have some examples. This makes the tool very easy to use. One specific example - The SRA Three Pass Protocol, is taken to explain the implementation, which makes it easier to understand. Each step of the protocol is explained in detail, and shows how this protocol can be modelled with the help of the update tool.

The update tool is mentioned in the coming section and then the rules are described one by one. Two new functions are implemented, namely Delete unreachable states and Normalizing state numbering. Whenever the rules are applied, there is a chance that some of the states in the output model becomes unreachable from the real world, and can be dropped. While drawing the models manually, these states are just not drawn. While doing it automatically, a function is made specifically to check for states that are unreachable and delete them in the output model. Once this is being done, the numbering of the states can turn out to be not sequential. So, another function (Normalize states) is made to make the state numbering sequential.

4.1 The Update tool - USSP (Update Semantics of Security Protocols)

A global function is made which is used to call all the definitions / methods and is very easy to use. Calling this function shows how to use all the definitions on input AUT files to get the corresponding output AUT files.

Giving "./ussp -h" as the input command shows the help file. It shows how to specify the input and output AUT files, how to specify which method should be performed, and also how to give the input parameters. There is also a debugging mode. That is the last option to be given in the input command. If no debugging is needed, the value 0 should be given as shown in the examples below. Value 1 is given for debugging the code.

```
[s031922@localhost addatom]$ ./ussp -h
Update semantics of security protocols: Version: 1.1
Usage: ussp -i <in> -o <out> [-a <AtomName>] [-f <Agent(s)>]
                             [-u <K,Agent(s)>]
                             [-s <Agent(s),Agent(s),K>]
                             [-d] [-h]

in                    : input file (.aut) <in> (default=./in.aut)
out                   : output file (.aut) <out>
AtomName              : The name of the atom to be added,
                      : example: K_n
Agent(s)              : Agents to be unfolded, example: "{(a),(b)}"
                      : OR "{(a,b),(c)}" ..
K,Agent(s)            : Update one or more agents with knowledge, K
Agent(s),Agent(s),K   : Side effect, example: "{(a),(b),(K_p)}"
                      : OR "{(a,b),(c),(K_p)}"..
                      : For example: "{K_p,b}" OR "{K_m,A}" ...
d                     : run ussp in debug mode
h                     : show this manual page
```

For example to do the ADDATOM function to an input file the command should be like:

```
./ussp -i in.aut -o out.aut -a "K_m" -d 0
```

-i is the code for giving the input AUT file, -o is the code for the giving the name of the output AUT file. -a is the code for ADDATOM.

To do the UPDATE functionality, the input command should be like:

```
./ussp -i in.aut -o out.aut -u "{K_ma,b}" -d 0
```

where, -u is the code for UPDATE functionality.

If UNFOLD functionality should be done for an input file:

```
./ussp -i in.aut -o out.aut -f "{(a),(b)}" -d 0
```

where -f is code for UNFOLD functionality.

To find out about the SIDE-EFFECT of agents, the command should be like:

```
./ussp -i in.aut -o out.aut -s "{(a),(b,c),(K_p)}" -d 0
```

where -s is the code for SIDE-EFFECT functionality.

Here the implementation methodology is explained by using an example - the SRA Three Pass Protocol. The protocol has the following steps:

1. $a \rightarrow b : \{x\}_{k_a}$
2. $b \rightarrow a : \{\{x\}_{k_a}\}_{k_b}$
3. $a \rightarrow b : \{x\}_{k_b}$

Here, both agent a and b have their own symmetric and unshared encryption key, k_a and k_b, respectively. Agent a sends a message x to agent b through an insecure channel after encrypting x. This is done by sending x protected with its own key. Next, b will encrypt this message with b's key and then sends this back to a. Since the encryption is assumed to be commutative, a can now decrypt this message and sends the result to b. Finally, b can decrypt the message it has just received and learn the value of x.

This protocol is modelled with three agents $\{a, b, c\}$. Agent c is supposed to be the intruder. It is assumed that all agents can see the activity of the network. In particular, they see messages been sent out and received. The interest is mainly in what agent c can learn during a run of this protocol between agents a and b. For modelling, some conventions are being used, for making the model easier to understand. Some useful propositions are defined. The propositions considered here are $\mathcal{P} = \{m, m_a, m_b, m_{ab}\}$ where m_a abbreviates $\{x\}_{k_a} = [\![\{x\}_{k_a}]\!]$ and m_{ab} abbreviates $\{\{x\}_{k_a}\}_{k_b} = [\![\{\{x\}_{k_a}\}_{k_b}]\!]$. Here $[\![y]\!]$ denotes the *real* value of y, i.e. the value of the expression y in the point of the model.

Here, the findings at the end of the protocol are
agent b believes m;
agent a learns that b believes m;
agent c learns only that a and b learns about m.

This protocol is captured by three public announcements. Since only public announcements, the step done in Update semantics is Update(ϕ,*). Starting state models the belief of agent a. Only agent a knows m and m_a. It is depicted in the figure 4.1. The state in square shows the starting state. The states in circles model the possible worlds. Transitions are shown with single sided / double sided arrows. The valuation corresponding to each states is shown inside the states (without the K_ prefix). The next sections in this chapter explains how the rules are being applied for a protocol. The picture shown in figure 4.1

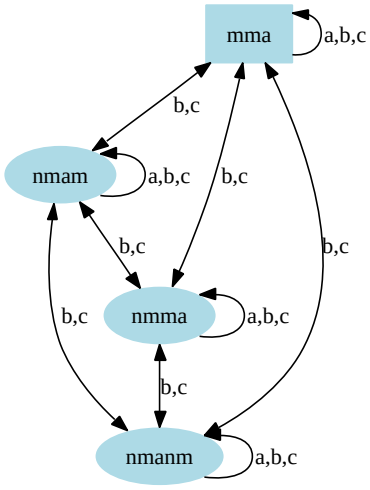

Figure 4.1: The initial state represented using the Update Semantics Model

is produced by one of the visualization tools. The visualization tool takes an AUT file as an input and produces a postscript (.ps) file with the image shown in the figure.

4.2 UPDATE

The first step of the protocol, i.e the public announcement of m_a, is executed. m_a is propagated on the network, so all agents will learn its value. So, we execute the action $Pub(a, m_a)$. The equivalent step in update semantics is Update(m_a,*).

So, the input command for the program tool should be

```
./ussp -i in.aut -o out.aut -u "{K_ma, A}" -d 0
```

The input command is parsed and the values K_ma and $\mathcal{A}$ are stored in buffers. With this information, the program knows K_ma is the knowledge that should be updated for all agents ($\mathcal{A}$ is the set of all agents)
Now, few values have to be set for the output list. Few formulas have been formed to set the values for the number of output states and transitions. When the states are duplicated, the output list will contain twice the number of states in the input list. The number of transitions in the output list can go up to 4 times the number of transitions in the input list.

Number of output states = (Number of input states * 2)
Number of output transitions = (Number of input transitions * 4)

These values are updated at the end of the function after actually creating all the transitions. So it is taken care that mistakes don't occur.

As mentioned before, both cases of update are dealt here, update for a specific set of agents OR update for all agents ($\mathcal{A}$). Consider the SRA Three Pass example.The input file given below represents the initial state of the protocol, where only agent a knows about m and m_a. The input list is given as an AUT file for the Kripke model that is shown in figure 4.1.

```
des(0,44,4)
(0,a,0)
(0,b,0)
(0,c,0)
(0,b,1)
(0,c,1)
(0,b,2)
(0,c,2)
(0,b,3)
(0,c,3)
(0,K_m,0)
(0,K_ma,0)
(1,a,1)
(1,b,1)
(1,c,1)
(1,b,0)
(1,c,0)
(1,b,2)
(1,c,2)
(1,b,3)
(1,c,3)
(1,K_nma,1)
(1,K_m,1)
(2,a,2)
(2,b,2)
(2,c,2)
(2,b,0)
(2,c,0)
(2,b,1)
(2,c,1)
(2,b,3)
(2,c,3)
(2,K_nm,2)
(2,K_ma,2)
(3,a,3)
(3,b,3)
(3,c,3)
```

```
(3,b,0)
(3,c,0)
(3,b,1)
(3,c,1)
(3,b,2)
(3,c,2)
(3,K_nma,3)
(3,K_nm,3)
```

The first line in the file gives three different kinds of information, the start state, number of transitions and number of states. The logic used for implementing the update function is as follows. From the first line in the input list, the number of input transitions and input states are taken and used to set the output transitions and states according to the equations above. Next step is to get the values from the input command and store them in buffers. The input values will have one (or more) update agents and one knowledge agent. Update agent can be just one specific agent or all agents ($\mathcal{A}$). These two cases are dealt separately.

Create new transitions. (For one specific agent)

Each transition is taken and the agent in the transition is compared with the update agent from the input command. If they are not the same, ignore it and take the next transition. If they are the same, then check if the knowledge of the destination state is the same as the knowledge agent from the input. If knowledge is not the same ignore it, and take the next transition. If they are the same, reproduce that specific transition in the output list also. this is done for all the transitions in the input list.

For all input transitions do
    If agent = agent in input command
    If destination knowledge = knowledge in input command
    Then
        Reproduce transition in output list

Here in the SRA Three Pass example, the update agent is $\mathcal{A}$, that means all agents are getting updated. Since this implementation is for one specific agent, this part of the code will not be encountered for this example.

Create old transitions. (For a specific agent)

Each transition in the input list is taken and reproduced in the output list after adding the number of input states to the source and destination states. For example, if input transition is (0, a, 0), then it is (3, a, 3) in the output list, if the number of input states is 3. This is done for all transitions in the input list.

For all input transitions do
source state = source state + number of input states
label = label
destination state = destination state + number of input states
Reproduce transition in output list

Create new to old transitions. (For a specific agent)

Each transition in the input list is taken and the agent in the transition is compared with the update agent from the input command. If they are not the same, reproduce it in the output list after adding the number of input states to the destination states. For example, if input transition is (0, a, 0), and "a" is not the update agent from the input command, then it is (0, a, 3) in the output list, if the number of input states is 3.

For all input transitions do
If agent != agent in input command
Then
source state = source state
destination state = destination state + number of input states
Reproduce transition in output list

Create new transitions. (For all agents $\mathcal{A}$)

Each transition is taken and the knowledge of the destination state is compared with the knowledge agent from the input command. If knowledge is not the same ignore it, and the next transition is taken. If they are the same, reproduce that specific transition in the output list also. This is done for all the transitions in the input list.

For all transitions do,
If destination knowledge = knowledge in input command
Then
Reproduce the transition.

This part of the code is encountered for the SRA Three Pass Protocol. A few transitions are taken from the example and the procedure is shown here. The knowledge agent to be checked is K_ma.

```
(0,a,0) = (0,a,0);      destination knowledge(0) = K_m, K_ma
(0,b,0) = (0,b,0);      destination knowledge(0) = K_m, K_ma
(0,c,0) = (0,c,0);      destination knowledge(0) = K_m, K_ma
(0,b,1) = do nothing;   destination knowledge(1) = K_m, K_nma
(0,c,1) = do nothing;   destination knowledge(1) = K_m, K_nma
(0,b,2) = (0,b,2);      destination knowledge(2) = K_nm, K_ma
(0,c,2) = (0,c, 2);     destination knowledge(2) = K_nm, K_ma
(0,b,3) = do nothing;   destination knowledge(3) = K_nm, K_nma
(0,c,3) = do nothing;   destination knowledge(3) = K_nm, K_nma
```

```
(0,K_m,0)  = (0,K_m,0)
(0,K_ma,0) = (0,K_ma,0)
```

Knowledge transitions are dealt separately. The valuations of the states never change. This procedure is done for all the remaining transitions.

Create old transitions. (For all agents $\mathcal{A}$)

Each transition is taken and reproduced in the output list after adding the number of input states to the source and destination states. For example, if input transition is (0, a, 0), then it is (3, a, 3) in the output list, if the number of input states is 3. This is done for all transitions in the input list.

Here in this example, the number of input states are 4. So, the transitions will change to:

```
(0,a,0) = (source + number of Input states, a,
        destination + number of Input states)
        = (0+4,a,0+4) = (4,a,4)
(0,b,0) = (0+4,b,0+4) = (4,b,4)
(0,c,0) = (0+4,c,0+4) = (4,c,4)
(0,b,1) = (0+4,b,1+4) = (4,b,5)
(0,c,1) = (0+4,c,1+4) = (4,c,5)
(0,b,2) = (0+4,b,2+4) = (4,b,6)
(0,c,2) = (0+4,c,2+4) = (4,c,6)
(0,b,3) = (0+4,b,3+4) = (4,b,7)
(0,c,3) = (0+4,c,3+4) = (4,c,7)

(0,K_m,0)  = (0+4,K_m,0+4) = (4,K_m,4)
(0,K_ma,0) = (0+4,K_ma,0+4) = (4,K_ma,4)
```

This procedure is done for all the remaining transitions. After applying any rule in the Update Semantics model, there can be states that are unreachable from the start state. These states should be discarded. Another separate function is written for deleting unreachable states.

4.3 Delete Unreachable States

For deleting the unreachable states, first the transitions from the starting state are checked. Two arrays are created as 'yes' array and the 'no' array.From the first line in the input list, the number of input transitions and input states are stored. All transitions from 0 (root) are taken, and the destination states are stored in a special array called the 'yes' array(Do not duplicate). This would mean that all the states that are in the 'yes' array are reachable from the root and they don't have to deleted from the list.

```
For all transitions from state 0 (root)
    yesarray[i] = destination state
```

Then the states that are not in the 'yes' array are stored in the 'no' array. If the 'no' array is empty, that means all states are reachable, and there is nothing to be deleted. So, exit.

```
For all states, do
    If state i is not in yesarray
        noarray[i] = state
```

Otherwise, take each state in the 'yes' array, and check its transitions. If there is transition from a state in the 'yes' array to a state in the 'no' array, that particular state in the 'no' array is shifted to the 'yes' array. This would mean that that particular state in the 'no' array is reachable from the root via some other state. So, that should not be deleted. If there is transition from a state in the 'yes' array to a state in the 'yes' array, then ignore it, do nothing. This procedure is done for all states in the 'yes' array, and also for the new states that are shifted to the 'yes' array from the 'no' array.

```
if noarray = empty, then exit
else
    For each state in yesarray,
        If transition exists from state 'i' in yesarray to state 'j' in noarray,
        Then
            yesarray[i] = state 'j'
```

In the end, if there are no states in the 'no' array that means there is nothing to be deleted. So, exit. Otherwise, if there are states present in 'no' array, then those states should be deleted. Also, all the transitions to and from those states should also be deleted.

```
if noarray = empty,
    exit
else
    states in noarray are not reproduced in the output list.
    transitions from and to these states are not reproduced in the output list.
```

Now, after deleting unreachable states (if there was any), there is one more problem. The numbering of the states does not have to be sequential anymore. So, the numbering of states has to be made sequential. Another function called NormalizeStates was written for handling this problem.

4.4 Normalizing State Numbering

All the states should be numbered sequentially. First set a variable state = 0. Each transition is checked if any one of them contains state 0 as the source or

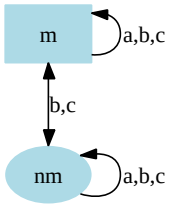

Figure 4.2: The Update Semantics Model after doing $Pub(a, m_a)$

destination state. Since 0 is the start state, it should be there. If it is found then the value of the state variable is incremented to 1 and start searching again. If the state is not found in any of the transitions, all the states with values higher than 0 is decremented by 1. This procedure is continued for all the transitions till the total number of states is checked. Now, the output list that we get is guaranteed to have the states numbered sequentially.

```
state = 0;
For all input transitions do
    If (source state = 0 OR destination state = 0) then
        state = state + 1;
    else
        Decrement the states with values higher than 0 by 1;
```

Now the whole update process is complete and the final model is as follows. It is shown in figure 4.2.

4.5 ADDATOM

Next step in the protocol is $Pub(b, m_{ab}))$. we notice that m_{ab} is not modelled yet, so this is the first thing to do. We will not repeat m_a in the figure since this holds in any state of the model. So, the next step is the operation $\text{ADDATOM}_{m_{ab}}$.

As mentioned in chapter 3, two copies have to be made from the input model. Each transition in the input model is converted to four transitions in the output model according to the algorithm given below. s and t are the states in the input model. s' and t' are the positive copies in the output model and s'' and t'' are the negative copies in the output model.

If s → t in the input model,
Then
    For all transitions except the knowledge transitions
    (in the output model),
        s' → t'
        s' → t"
        s" → t'
        s" → t"

Hence the number of transitions in the output list is the number of transitions in the input list multiplied by four. And since two copies if the input model are made, the number of states in the output model is the number of input states multiplied by two. While implementing, each transition is taken and it is transformed to four transitions as mentioned before. But, all knowledge transitions are taken care of separately. They are just transformed as it is in the input model, and also the new value is added.

The input list for doing ADDATOM for the next step of the protocol is given below. This is the LTS form for the figure 4.2.

```
des(0,12,2)
(0,a,0)
(0,b,0)
(0,b,1)
(0,c,0)
(0,c,1)
(0,K_m,0)
(1,a,1)
(1,b,0)
(1,b,1)
(1,c,0)
(1,c,1)
(1,K_nm,1)
```

Each transition is taken and is transformed to four transitions. 0 and 1 are the states in the input list. So, the states in output list will be 0, 1, 2, and 3. According to the pseudo code above, if s and t are 0 and 1, then s' and t' are 0 and 1, and s'' and t'' are 2, and 3.

```
(0,a,0) = (0,a,0)
        = (0,a,2)
        = (2,a,0)
        = (2,a,2)

(0,b,0) = (0,b,0)
        = (0,b,2)
```

```
        = (2,b,0)
        = (2,b,2)

(0,b,1) = (0,b,1)
        = (0,b,3)
        = (2,b,1)
        = (2,b,3)

(0,c,0) = (0,c,0)
        = (0,c,2)
        = (2,c,0)
        = (2,c,2)

(0,c,1) = (0,c,1)
        = (0,c,3)
        = (2,c,1)
        = (2,c,3)

(0,K_m,0) = (0,K_m,0)
          = (2,K_m,2)
```

Like this all transitions are transformed. Also, the new atom atom is added to all the four states. As mentioned in section ,0 and 1 are the positive states and 2 and 3 are the negative states. So, the value of the new atom added is true in the positive states and false in the negative states.

```
(0,K_mab,0)
(1,K_mab,1)
(2,K_nmab,2)
(3,K_nmab,3)
```

After doing ADDATOM, the result is shown in figure 4.3. As mentioned before, the next step in the protocol is.$Pub(b, m_{ab})$) The equivalent step in update semantics is Update(m_a,*).This is shown in figure 4.4.

The third step in the protocol is $Pub(b, m_b)$). Since m_b is not present in the model, the equivalent steps in Update semantics are ADDATOM_{m_b} and Update(m_b,*). m_a will not be repeated in the figure since this holds in any state of the model.These are shown in figures 4.5 and 4.6 respectively.

Now the next step is to find out how much each agents have learnt. Basically, next step is to prove the three findings mentioned above. m_b will not be repeated in the figure since this holds in any state of the model. It has to be found out whether agent b learns m, and also what agent c has learned. So, the equivalent steps in update semantics are Update(m_a,b) and $S_{*,b}m$ (side-effect that everyone learns that agent b learns m). The figure after doing Update(m_a,b) is

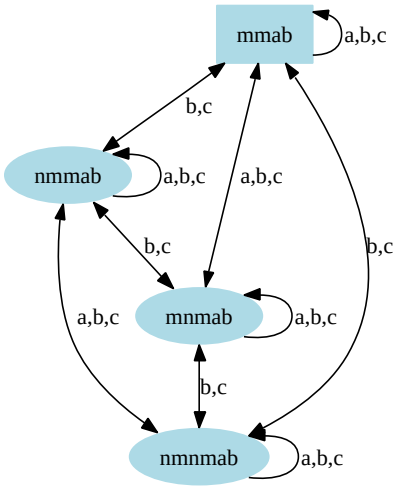

Figure 4.3: The Update Semantics Model after doing ADDATOM$_{m_{ab}}$

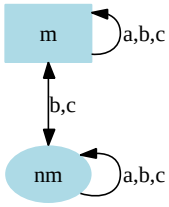

Figure 4.4: The Update Semantics Model after doing $Pub(b, m_{ab})$)

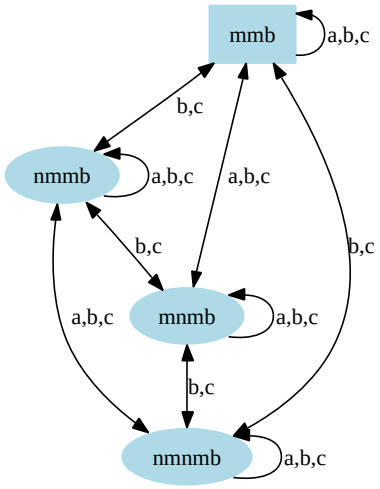

Figure 4.5: The Update Semantics Model after doing ADDATOM$_{m_b}$

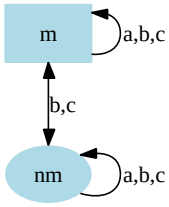

Figure 4.6: The Update Semantics Model after doing $Pub(b, m_b)$)

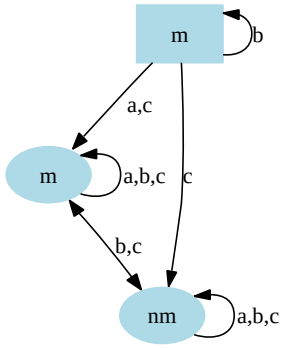

Figure 4.7: The Update Semantics Model after doing Update(m_a,b)

shown in figure 4.7.

4.6 SIDE-EFFECT

The final step in the protocol is to find the side-effect of agents. The input command to be given to the program tool is

```
./ussp -i in.aut -o out.aut -s "{(a,c),(b), (K_m)}" -d 0
```

This command finds out the side-effects of agents a and c, with respect to agent b learning K_m. As mentioned in section 3.1.2 in chapter 3, several subtransformations have to be performed in this procedure, and they are unfolding, finding the partial submodel and then applying ATOMSPLIT.

4.6.1 Unfold

In this section, implementing Unfold is explained. What should be done is $\text{UNFOLD}_{\{(a,c),(b)\}}$. One extra feature of the tools is unfolding can be done separately without doing the side-effect function. This is just to get an idea how the knowledge is being separated. The input command to do only unfolding for the program tool should be

```
./ussp -i in.aut -o out.aut -f "{(a,c),(b)}" -d 0
```

The input command is parsed and the values (a,c) and (b) is stored in buffers. This is the information needed for unfolding the input list. (a,c) is taken as partition 1 and (b) is taken as partition 2. If there are n partitions, then the transitions in the input list are duplicated n times, plus the transitions from the root for each agents are added extra for the specific partitions. Its almost similar for the calculation for the output states also. If the input model has n

partitions, then the output model will have n times the number of input states, plus one new starting point. Two equations are formed for calculating the output states and transitions according to the logic mentioned above.

Number of output states = 1 + (number of input states) * (number of partitions).
Number of output transitions = count + (number of input transitions * number of partitions).
where count is just the number of transitions from the root in the input list.

Consider the model. The input list for this model is given below. And this is the AUT file for the model shown in figure 4.7.

```
des(0,17,3)
(0,a,1)
(0,b,0)
(0,c,1)
(0,c,2)
(0,K_m,0)
(1,a,1)
(1,b,1)
(1,b,2)
(1,c,1)
(1,c,2)
(1,K_m,1)
(2,a,2)
(2,b,1)
(2,b,2)
(2,c,1)
(2,c,2)
(2,K_nm,2)
```

As can be seen, the first line gives the three different vital information. It says des(0,17,3), where 0 is the start state, 17 is the number of transitions and 3 is the number of states. Rest of the lines represents the 17 transitions in the model. While implementing, the first line is used to retrieve the vital information mentioned above. As mentioned before, 0(zero) is always the root in all the models.

Creating new transitions

Input states have to be duplicated for each partition. Root is $s0$ in the input list. Root is $s'0$ in the output list. A constant is set to the value 1; const = 1. This is used for changing the state numbering in each of the partitions. If there is a transition that's a self loop from the root, the transition goes to the duplicate state for root for that particular partition. If $(0, a, 0)$, and a is in the

first partition, it becomes $(0, a, 1)$, destination = destination + const. Increment const for the next partition. See how many partitions are there and make that much duplication of the states and transitions.

Const = 1;
Take each transition and do;
    For each partition;
        Source state = source state + const;
        Label = label;
        Destination state = destination state + const;
    Const = const + number of input states;

As mentioned before, (a,c) is taken as partition 1 and (b) is taken as partition 2. First take the transitions from the root. They are

```
(0,b,0)
(0,K_m,0)
(0,a,1)
(0,c,1)
(0,c,2)
```

Set const = 1.

```
(0,b,0)   = do nothing; since b is not there in partition 1.
(0,a,1)   = (0,a,2) ; destination = destination + const
                    ; since a is in partition 1.
(0,c,1)   = (0,c,2) ; destination = destination + const
                    ; since c is in partition 1.
(0,c,2)   = (0,c,3) ; destination = destination + const
                    ; since c is in partition 1.

(0,K_m,0) = (0,K_m,0)
```

Knowledge remains the same for the root in the output list also.
Set const = 1 + number of input states = 1 + 3 = 4. (For partition 2)

```
(0,b,0) = (0,b,4) ; destination = destination + const ;
        since b is in partition 2.
(0,a,1) = do nothing;
        since 'a' is not there in partition 2.
```

since only (b) is in partition 2, all the other transitions have no effect. So, now the transitions from the new starting point is made.Since there are two partitions, all the input transitions should be duplicated twice. The first duplication is for the first partition, and the same for the second.

```
(0,a,1)     = (1,a,2);
            (0 +  const = 0 + 1 = 1); (1 +  const = 1 + 1 = 2)
```

```
            = (4,a,5);
            (0 +  (const + Input states) = 0 + (1 + 3) = 4);
            (1 +  (const + Input states) = 1 + (1 + 3) = 5);

(0,b,0)     = (1,b,1);
            (0 +  const = 0 + 1 = 1); (0 +  const = 0 + 1 = 1);
            = (4,b,4);
            (0 +  (const + Input states) = 0 + (1 + 3) = 4);
            (0 +  (const + Input states) = 0 + (1 + 3) = 4);

(0,c,1)     = (1,c,2);
            = (4,c,5);

(0,c,2)     = (1,c,3);
            = (4,c,6);

(0,K_m,0)   = (1,K_m,1);
            = (4,K_m,4);

(1,a,1)     = (2,a,2);
            = (5,a,5);

(1,b,1)     = (2,b,2);
            = (5,b,5);

(1,b,2)     = (2,b,3);
            = (5,b,6);

(1,c,1)     = (2,c,2);
            = (5,c,5);

(1,c,2)     = (2,c,3);
            = (5,c,6);

(1,K_m,1)   = (2,K_m,2);
            = (5,K_m,5);

(2,a,2)     = (3,a,3);
            = (6,a,6);

(2,b,1)     = (3,b,2);
            = (6,b,5);

(2,b,2)     = (3,b,3);
            = (6,b,6);
```

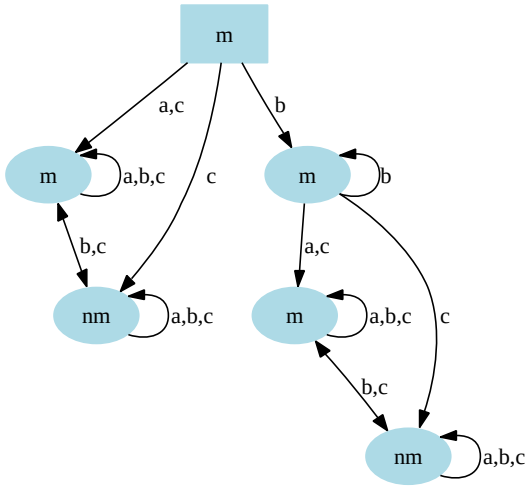

Figure 4.8: The Update Semantics Model after doing UNFOLD

```
(2,c,1)      = (3,c,2);
             = (6,c,5);

(2,c,2)      = (3,c,3);
             = (6,c,6);

(2,K_nm,2)   = (6,K_nm,6);
             = (6,K_nm,6);
```

The picture showing the unfolded model is given in figure 4.8. It can be seen in the picture that state 6 has been deleted from the figure, because it is unreachable from the root (starting state). Always after every operation, the model is checked for unreachable states.

4.6.2 Final steps of side-effect

Next step in the process to find the submodel, and then applying ATOMSPLIT in that partial submodel. This is done while unfolding itself. The implementation of UNFOLD is in such a away that, the agents whose side-effect has to be learnt will always be partition 1. So, that means, the partial submodel contains all the transitions and states specific for partition 1, the transitions from the starting state and the starting state itself. During implementation ATOMSPLIT is applied to only these transitions.

```
des(0,34,6)
(0,a,1)
(0,b,3)
```

```
(0,c,1)
(0,c,2)
(0,K_m,0)
(1,a,1)
(1,b,1)
(1,b,2)
(1,c,1)
(1,c,2)
(1,K_m,1)
(2,a,2)
(2,b,1)
(2,b,2)
(2,c,1)
(2,c,2)
(2,K_nm,2)
(3,a,4)
(3,b,3)
(3,c,4)
(3,c,5)
(3,K_m,3)
(4,a,4)
(4,b,4)
(4,b,5)
(4,c,4)
(4,c,5)
(4,K_m,4)
(5,a,5)
(5,b,4)
(5,b,5)
(5,c,4)
(5,c,5)
(5,K_nm,5)
```

This is the input list for the model in figure 4.8. So, the states numbering 0,1 and 2 are contained in the partial submodel, and ATOMSPLIT is applied to them. As mentioned in chapter 3, ATOMSPLIT is applied to all transitions having label as agent *b*.

```
For each transition do;
If label = b
then
    If source knowledge == K_m
    then
        knowledge = K_m;
    else
        knowledge = K_nm;

    If destination knowledge == knowledge
    then
        reproduce the transition in the output list;
    else
        delete the transition;
else
    Do nothing;
```

```
(0,a,1) = label is a = do nothing, and
        simply reproduce the transition in the output list.

(0,b,3) = source knowledge = destination knowledge = K_m;
        simply reproduce the transition in the output list.

(0,c,1) = label is c = do nothing, and
        simply reproduce the transition in the output list.

(0,c,2) = label is c = do nothing, and
        simply reproduce the transition in the output list.

(0,K_m,0) = do nothing, and
          simply reproduce the transition in the output list.

(1,a,1) = label is a = do nothing, and
        simply reproduce the transition in the output list.

(1,b,1) = source knowledge = destination knowledge = K_m;
        simply reproduce the transition in the output list.

(1,b,2) = source knowledge = K_m and destination
        knowledge = K_nm; so, delete this transition.

(1,c,1) = label is c = do nothing, and
        simply reproduce the transition in the output list.

(1,c,2) = label is c = do nothing, and
        simply reproduce the transition in the output list.
```

```
(1,K_m,1) = do nothing, and
          simply reproduce the transition in the output list.

(2,a,2) = label is a = do nothing, and
        simply reproduce the transition in the output list.

(2,b,1) = source knowledge = K_nm and destination
        knowledge = K_m; so, delete this transition.

(2,b,2) = source knowledge = destination knowledge = K_nm;
        simply reproduce the transition in the output list.

(2,c,1) = label is c = do nothing, and
        simply reproduce the transition in the output list.

(2,c,2) = label is c = do nothing, and
        simply reproduce the transition in the output list.

(2,K_nm,2) = do nothing, and
           simply reproduce the transition in the output list.
```

Now the whole process of applying side-effect is over. And the final picture is given in figure 4.9. From this figure, the three initial findings can be verified. That is,
agent b believes m;
agent a learns that b believes m;
agent c learns only that a and b learns about m.

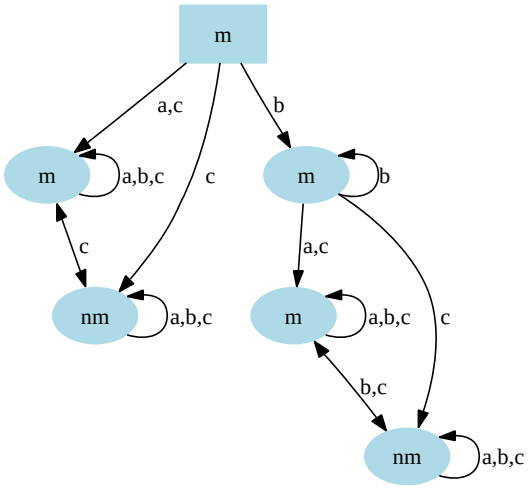

Figure 4.9: The final model after doing the side-effect function

Chapter 5

Visualization and verification of Security Protocols using USSP and CADP

This chapter is mainly about how the update tool can be used for the visualization and verification of security protocols. All the output files that are generated by the update tool are in AUT format, which is well accepted by the CADP model checker. It must be clear by now that visualization is very much possible after seeing the figures in the previous chapters.

The first section in this chapter explains how the CADP model checker helps in verifying the belief level of each agents that are involved. The next section takes the famous Needham Schroeder Key Exchange protocol and explains the whole procedure step by step. First, the protocol is explained and then modelling it with Update Semantics is shown. Each stage is visualized with the help of an existing tool and then the protocol is verified with the CADP model checker. The third section deals with yet another famous protocol, the Wide-Mouthed Frog protocol.

5.1 Verifying epistemic formulas with CADP

The knowledge of agents can be automatically verified using CADP model Checker. This CADP model Checker accepts the AUT files as input. It takes two inputs for verification. One is the AUT file and the next is the mu-calculus file having the formula for checking the knowledge. While each step of the protocol is modelled using update tools, the belief level of all the participating agents are verified. Consider as an example, one step of a protocol:

$a \rightarrow b : \{m\}pkb$

While this is being modelled using update semantics, questions that can arise about the knowledge of both the agents a and b are:

Does agent a believes m ?
Does agent a believes $\{m\}pkb$?
Does agent b believes m ?
Does agent b believes $\{m\}pkb$?
Does agent c believes m ?
Does agent c believes $\{m\}pkb$?
Does agent c learn that a and b believes m ?

These questions can be easily verified using the CADP model checker. Specific formulas can be formulated for all these questions to find the answers.

Consider an example.

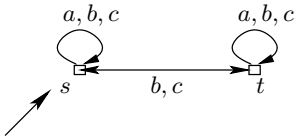

Figure 5.1: (M, s)

The valuation for the states are given as $\pi(s)(p) =$ `true`, $\pi(t)(p) =$ `false`. It can be seen from the figure that only agent a believes p ($B_a p$). This can be verified using the model checker. The formulas are also in the text format and are saved as .mcl files. The specific formulas for verifying the knowledge of each agent is given below.

agent a believes p

```
["a"]<"K_p">true - TRUE
```

(This means that - For agent a, in all worlds visible by a from the real world, p is true. Then it proves that agent a knows the value of p).

agent b believes p

```
["b"]<"K_p">true - FALSE
```

agent c believes p

```
["c"]<"K_p">true - FALSE
```

It can be seen that it is very easy to verify using the model checker. This comes in handy when the model is too big and crowded. This will be again explained more in the next section using an example.

5.2 The Needham Schroeder Key Exchange Protocol

Few examples are taken to show how much the update tool is useful in modelling the security protocols, and then using the output to visualize and verify them. The update tool has been used for several protocols to analyse the results.Here the update tool was applied for Needham Schroeder Key exchange Protocol. and in chapter 4 the tool was applied to SRA Three Pass Protocol.

The Needham-Schroeder Key Exchange Protocol is a typical example that has been widely examined by protocol researchers. Users are s(sender) and r(receiver). s knows m, secret key of s, sks and public key of s, pkr. r knows p, secret key of r, skr and public key of s, pks. m and p are secret key values of s and r respectively. $\{x\}pkr$ denotes encryption of the value x with the public key of r. The protocol has the following steps.

- $s \rightarrow r : \{s, m\}pkr$
- $r \rightarrow s : \{m, p\}pks$
- $s \rightarrow r : \{p\}pkr$

Initially s (sender) sends its identity and secret key value to r (receiver), after encrypting with the public key of r. Then, r decrypts it with its own secret key and reads the content of the message. Then r sends its own secret key along with the sender's secret key back to s, encrypted with the public key of s. r sends back the secret key of s, showing that r did receive it and so s can be sure that the previous message was received by the right person. And r, by sending its own secret key, authenticates itself.

Conventions used while modelling
Several conventions are used to make the modelling easy and helps in understanding the model better. All the messages in the protocol are given some abbreviations to make it easy to model with Update Semantics.

m is taken as the secret key of s. And p is taken as the secret key of r.
$m_a = \{s, m\}pkr$,
$m_b = \{p\}pkr$,
$m_{ab} = \{m, p\}pks$.

Initial Knowledge
Initially s knows its own secret key value m, its own private key sks and the public key of r, pkr. And r knows its own secret key value p, its own private key skr and and the public key of s, pks.

Modelling using Update Semantics
Here this protocol is modelled using Update semantics. In this modelling, three agents $\{s, r, c\}$ are considered. c is supposed to be the Intruder. It is assumed

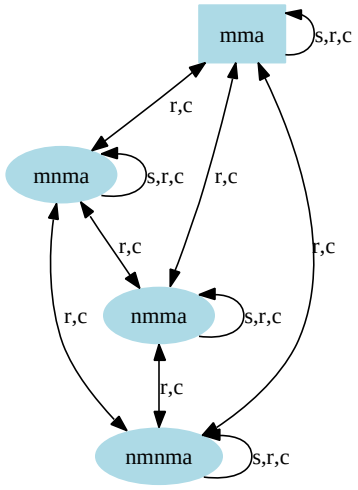

Figure 5.2: The initial state represented using the Update Semantics Model

that all agents can see the activity of the network. In particular, they see messages been sent out and received. Here the interest is mainly in knowing what each agents learn during a run of this protocol between agents s and r.

First the steps of the protocol are converted to the Update Semantics model. The initial state is represented in Figure 5.2 and Figure 5.3. The starting state shows the real world and all other are possible worlds for the agents. s, r and c are agents. The valuations for each state are given inside the states (without the "K_" prefix). And transitions shows what each agent considers possible.

The initial model only shows the initial knowledge of agent s. For agent s, from the real world, it sees K_m in all the possible worlds. This can be seen from the picture given in Figure 5.2 and Figure 5.3. So, it can be stated that agent s believes m ($B_s m$). The belief formulas are explained in section 2.1. The agents r and c has got transitions to different states where there are different valuations for m. That means they are not sure of the value of m. So, it can be said that $\neg B_r m$ (r does not know the value of m) and $\neg B_c m$ (c does not know the value of m).

While modelling the initial state, it is already assumed that agent s has already prepared the first encrypted message m_a. So the initial model consists of propositions m and m_a. Also, agent s initially knows its own private key sks and the public key of r, pkr. These were deliberately left out of the modelling because we don't model the encryption/decryption process.

Now the first step of the protocol is executed. The equivalent steps in update

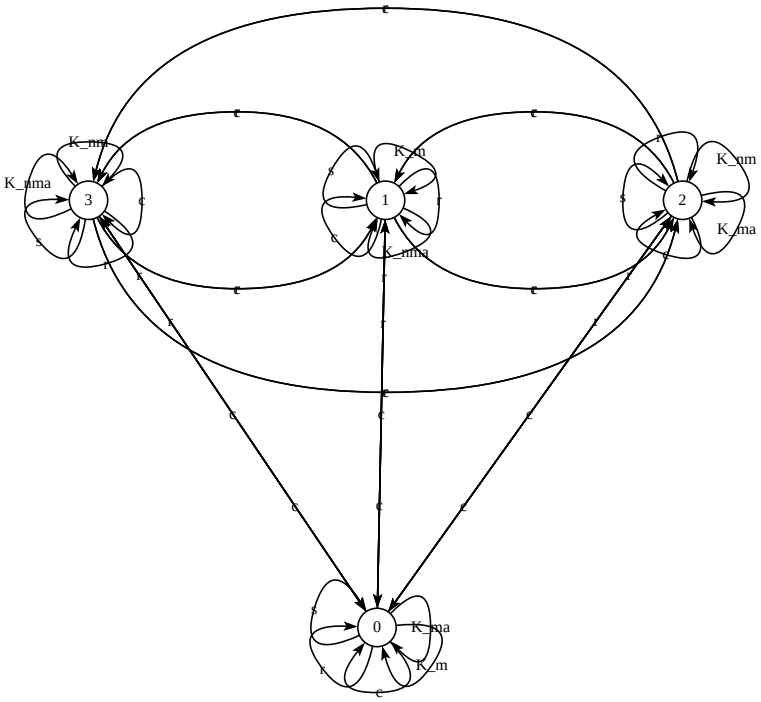

Figure 5.3: The initial state represented using the Update Semantics Model

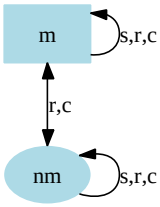

Figure 5.4: After doing UPDATE$_{(m_a,*)}$

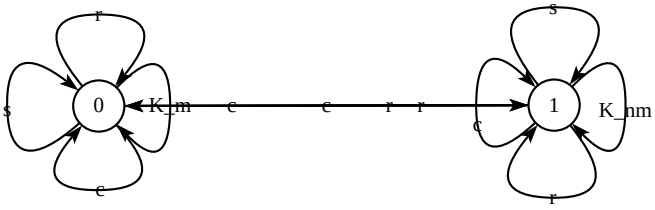

Figure 5.5: After doing UPDATE$_{(m_a,*)}$

semantics are UPDATE$_{(m_a,*)}$ and UPDATE$_{(m,r)}$. The models are shown in figures 5.4 and 5.6 respectively. The models are also shown in figures 5.5 and 5.7 using the visualization tool in CADP [FGK+96]. It can be seen that m_a is not shown in the pictures. Since every agent learns about it, its value is the same everywhere. So, it is removed to make the model easier.

Next comes the second step in the protocol. The message $m_{ab} = \{m, p\}pks$ is sent by agent r to s. Since p and m_{ab} are not in the model yet, they have to added first before modelling this step. After adding them, according to the modelling methodology mentioned in section 3.2, update steps and side-effect

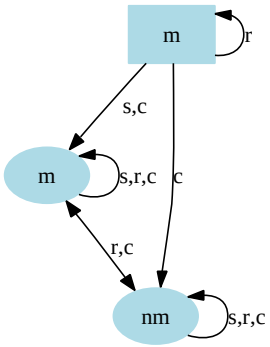

Figure 5.6: After doing UPDATE$_{(m,r)}$

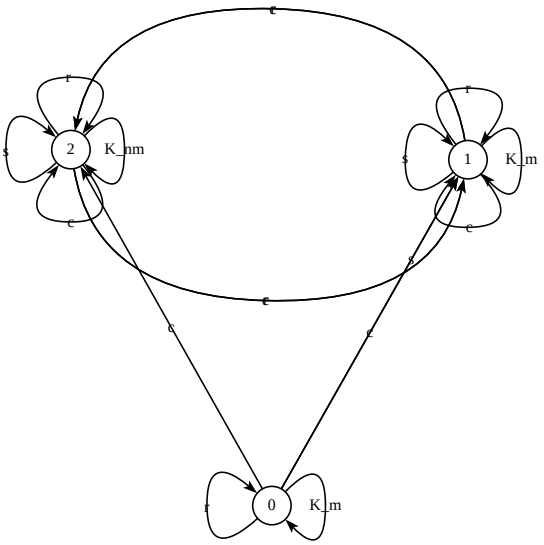

Figure 5.7: After doing $\text{UPDATE}_{(m_a,r)}$

steps are performed. The total number of steps involved in this protocol are:

ADDATOM_{p},
$\text{ADDATOM}_{m_{ab}}$,
$\text{UPDATE}_{(m_{ab},*)}$,
$\text{UPDATE}_{(p,s)}$,
$\text{SIDE-EFFECT}_{(m,s,r)}$.

The final model after applying all these steps are given in Figure 5.8 and Figure 5.9 using two different visualization tools. After the second step, the knowledge to be verified are:

$B_r m$ - agent r believes m
$B_s p$ - agent s believes p
$B_s B_r m$ - agent s learns that agent r believes m
$\neg B_r B_s p$ - agent r does not learn that agent s believes p

Automatic verification of these final findings using the CADP model checker is given later on in this chapter.

Now the last and final step of the protocol is modelled. The message $m_b = \{p\}pkr$ is sent by agent s to agent r. Since m_b is not in the model, it has to be added as before. And then the update steps are performed. The steps involved in this protocol are:

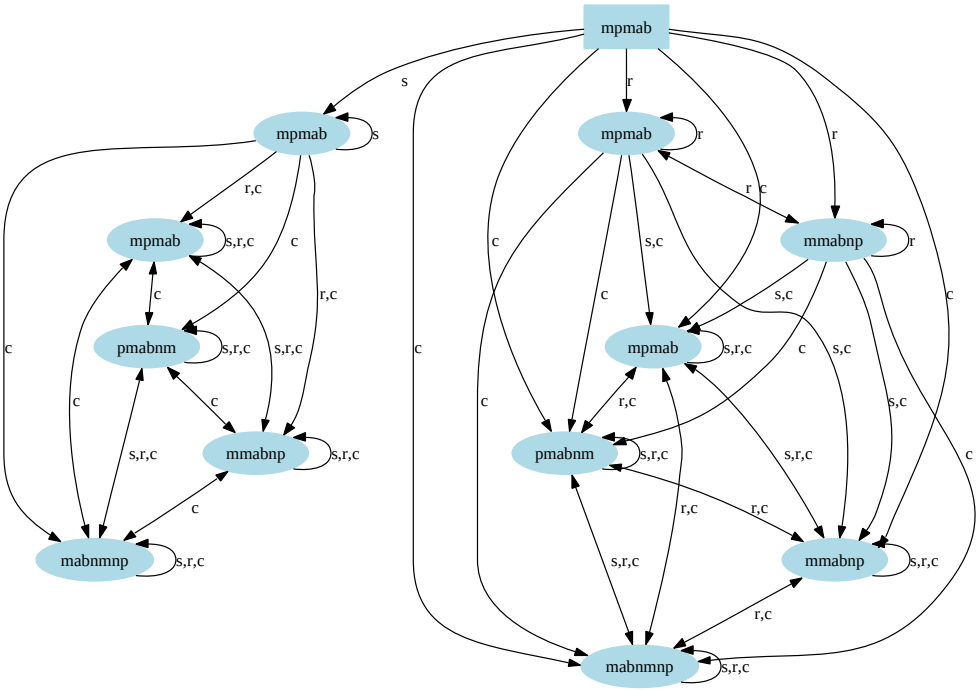

Figure 5.8: After the second step of the protocol is modelled

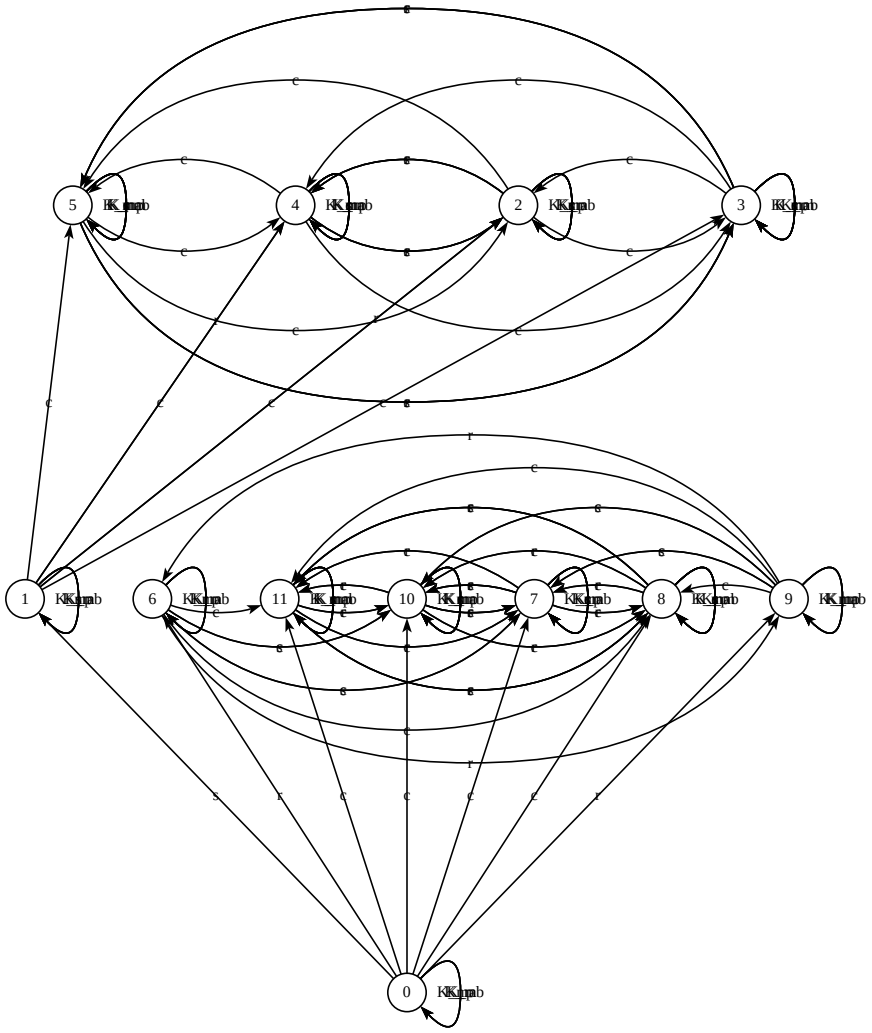

Figure 5.9: After the second step of the protocol is modelled

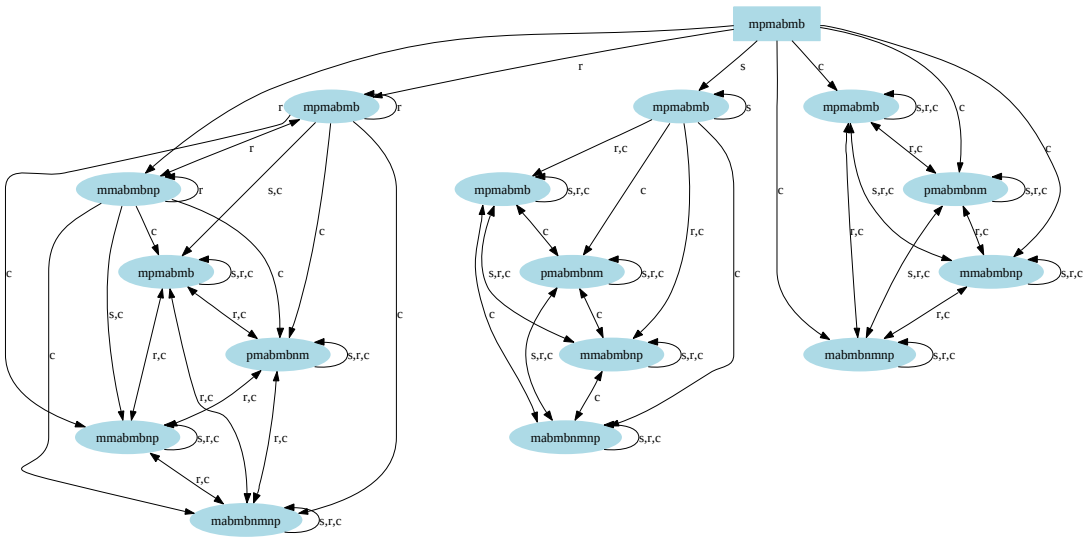

Figure 5.10: At the end of the protocol

ADDATOM$_{m_b}$,
UPDATE$_{(m_b,*)}$,
SIDE-EFFECT$_{(p,r,s)}$.

The final model after applying all these steps are given in figures 5.10 and 5.11. At the end of the protocol, the questions that can be answered are:

$B_s m$ - agent s believes m
$B_s p$ - agent s believes p
$B_r m$ - agent r believes m
$B_r p$ - agent r believes p
$B_s B_r m$ - agent s learns that agent r believes m
$B_r B_s p$ - agent r learns that agent s believes p
$\neg B_c m$ - agent c does not learn about m
$\neg B_c p$ - agent c does not learn about p

Automatic verification of these findings using the CADP model checker is explained in the section below. As you can see, the picture becomes bigger and messy as the protocol progresses. So, it is very easy to make mistakes. This shows how much the tool is useful in generating these big models instantly.

5.2.1 Knowledge Verification using CADP

In this section each step of the Needham Schroeder protocol is revisited. First we take the initial state of the protocol that is shown in figure 5.2. In every

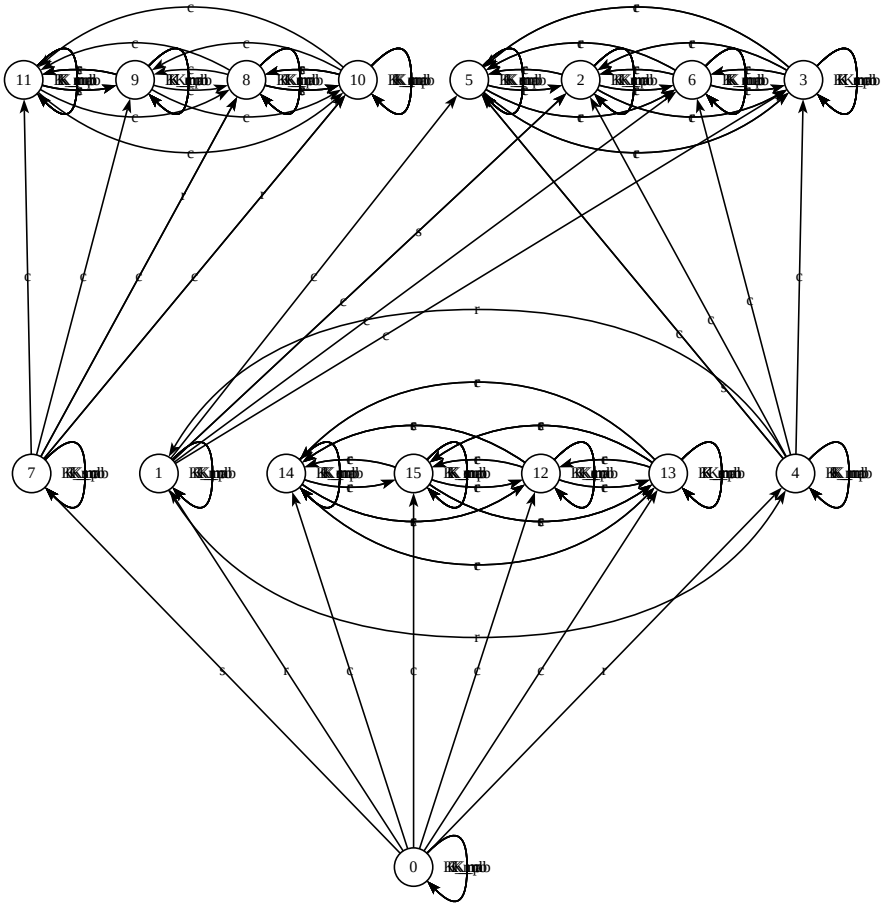

Figure 5.11: At the end of the protocol

stage, the knowledge of all the three agents can be checked.

Initial Model

First the initial model is taken that is shown in figure 5.2. Since it shows the initial knowledge of agent s, it can be checked what the agents r and c knows about that.

agent s believes m

```
["s"]<"K_m">true - TRUE
```

agent r believes m

```
["r"]<"K_m">true - FALSE
```

agent c believes m

```
["c"]<"K_m">true - FALSE
```

For s knows m, the model checker returns TRUE. But, for agent r knows m and for agent c knows m, the model checker returns FALSE. I think that is obvious from the figure itself. But to make sure the figure is right, the check can be done. Check can also be done this way:

agent c does not learn about m

```
<"c">["K_m"]false - TRUE
```

This means that agent c considers m and $\neg m$ possible. The model checker returns TRUE, which shows that agent c does not know the actual value of m. In this way, for every step of the protocol the verification can be done.

After the first step of the protocol

After the first step is completely modelled with the update tool,again, verification is done for the finding out how much each agent has learnt. Agent r has learnt the message m, that should be the only knowledge update after the first step of the protocol.

agent s believes m

```
["s"]<"K_m">true - TRUE
```

agent r believes m

```
["r"]<"K_m">true - TRUE
```

agent c believes m

```
["c"]<"K_m">true - FALSE
```

After the second step of the protocol

After the second step is completely modelled with the update tool,again, verification is done for the finding out how much each agent has learnt. Agent s has learnt the message p, and agent s knows that agent r has learnt the message m. These are the knowledge updates after the second step of the protocol.

agent s believes m

```
["s"]<"K_m">true - TRUE
```

agent r believes m

```
["r"]<"K_m">true - TRUE
```

agent c believes m

```
["c"]<"K_m">true - FALSE
```

agent s believes p

```
["s"]<"K_p">true - TRUE
```

agent r believes p

```
["r"]<"K_p">true - TRUE
```

agent c believes p

```
["c"]<"K_p">true - FALSE
```

After the third and final step of the protocol

After the third and final step of the protocol is completely modelled with the update tool, verification is done as before. Now agent r knows that agent s has learnt the message p, and all other knowledge updates remains the same.

agent s believes m

```
["s"]<"K_m">true - TRUE
```

agent r believes m

```
["r"]<"K_m">true - TRUE
```

agent c believes m

```
["c"]<"K_m">true - FALSE
```

agent s believes p

```
["s"]<"K_p">true - TRUE
```

agent r believes p

```
["r"]<"K_p">true - TRUE
```

agent c believes p

```
["c"]<"K_p">true - FALSE
```

5.2.2 Attack on this protocol

To find an attack, all the traces should be taken and modelled. Here, we can model traces but not generate them automatically. The attack for this famous Needham Schroeder Key exchange protocol was found out in 1995, which was introduced in 1978. The trace that shows the attack is given below.

$s \rightarrow c : \{s, m\}pkc$	$c \rightarrow r : \{s, m\}pkr$
$r \rightarrow c : \{m, p\}pks$	$c \rightarrow s : \{m, p\}pks$
$s \rightarrow c : \{p\}pkc$	$c \rightarrow r : \{p\}pkr$

As mentioned before, the intention is not to prove that the protocol is completely secure. It is mainly about learning the knowledge of the agents involved, mainly the intruder's knowledge and how it gets updated. Even if the protocol that is modelled is not secure, it cannot be found out by just taking one trace of the protocol. Here, in update semantics, at a time, only one trace of the protocol is considered. To find an attack, all the traces should be taken and modelled.

5.3 The Wide-Mouthed Frog protocol

Another example is also used to illustrate the power of the update tool, that is the well-known Wide-Mouthed Frog protocol (see,[BAN96]). The protocol exchanges a session key k from the agent a to another agent b via a server s. Then, agent a sends agent b a message protected with the session key k. It is assumed, that the agents a and b share each a symmetric key, k_{as} and k_{bs} say, with the server. The protocol is described below.

1. $a \rightarrow s : \{k\}_{k_{as}}$
2. $s \rightarrow b : \{k\}_{k_{bs}}$
3. $a \rightarrow b : \{m\}_k$

The keys k_{as} and k_{bs} are shared among a and s, and among b and s, respectively. The key k is fresh and initially only known to agent a, as is message m. In the analysis focus is on the session key k and the message m it protects. The security of the channel, based on the server keys k_{as} and k_{bs} is expressed by private rather than public communication. It is assumed that the 'ports' of the channel from a to b can be observed, but the ones for the communication with the server are not visible to other parties.

The initial knowledge is depicted in Figure 5.12. Execution of the first step of the protocol leads to an update of the knowledge of s. This is represented by the model in the Figure 5.13. Similarly, the execution of the second step of the protocol induces the model in Figure 5.14. The model becomes bigger due to the assumption that the learning of messages exchanged with the server

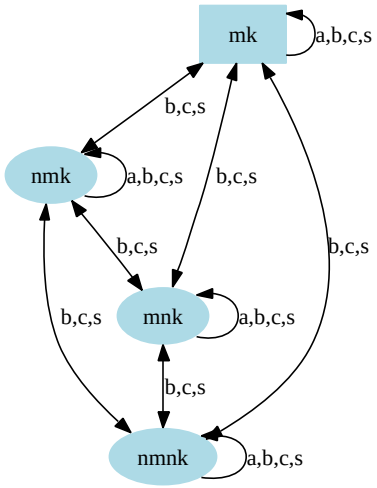

Figure 5.12: The initial state of the protocol.

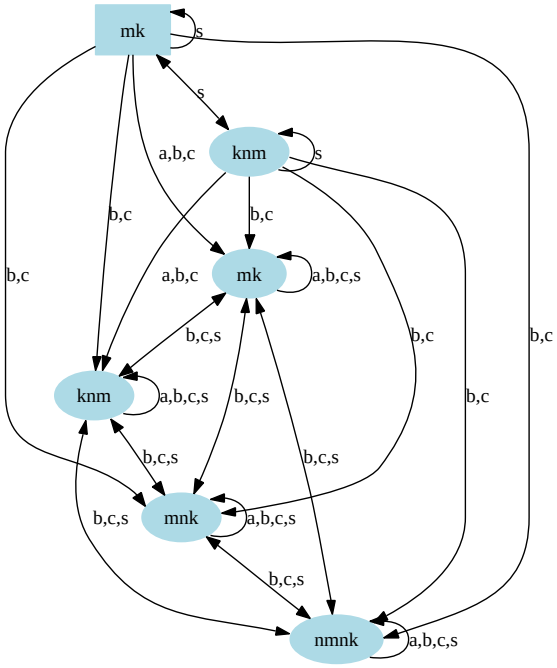

Figure 5.13: After the execution of the first step of the protocol.

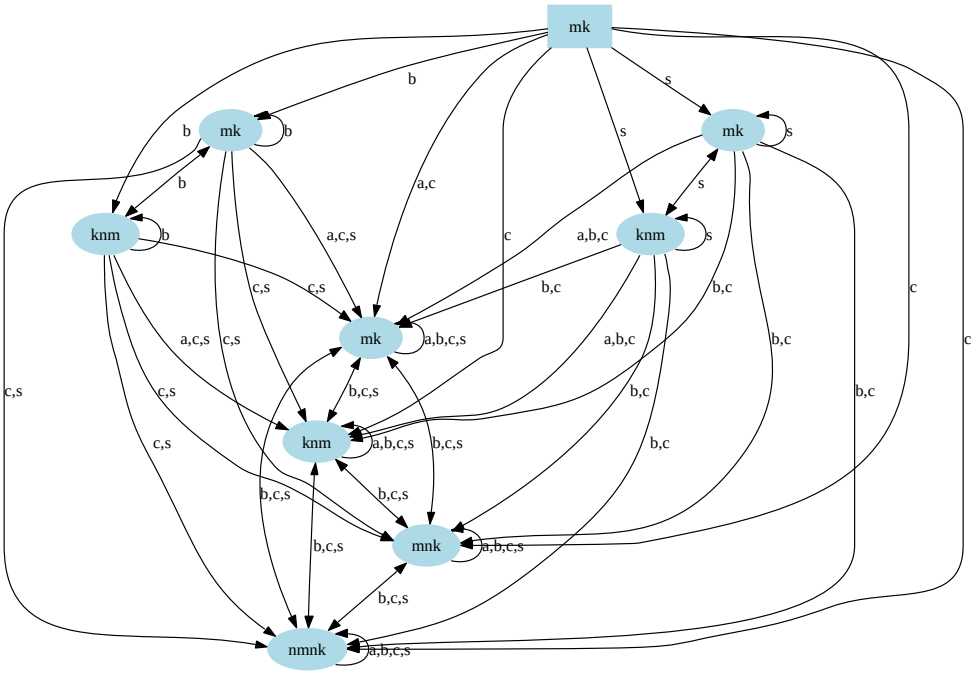

Figure 5.14: After the execution of the second step of the protocol.

is private. For example, agent a is not aware of agent b learning about the key k.

Finally, the last step of the protocol is executed. It is the public communication of the message m_k. For this we need to add the atom m_k abbreviating $\{m\}_k = [\![\{m\}_k]\!]$. This doubles the number of states of the models. However, since m_k will be known to all agents, its negative part can be discarded. So, agent b learns the content m and the other agents learn that b knows about m and k. The Figure 5.15, shows the learning of content m by agent b. The findings at the end of the protocol are $B_b(m \wedge k)$, agent b knows the values of the message m and session key k, and, $\neg B_c B_s k$ agent c does not know that the server knows the session key k.

5.3.1 Knowledge Verification using CADP

In this section each step of the protocol is revisited. First we take the initial state of the protocol that is shown in figure 5.12. In every stage, the knowledge of all the three agents are checked with respect to message m and the key k.

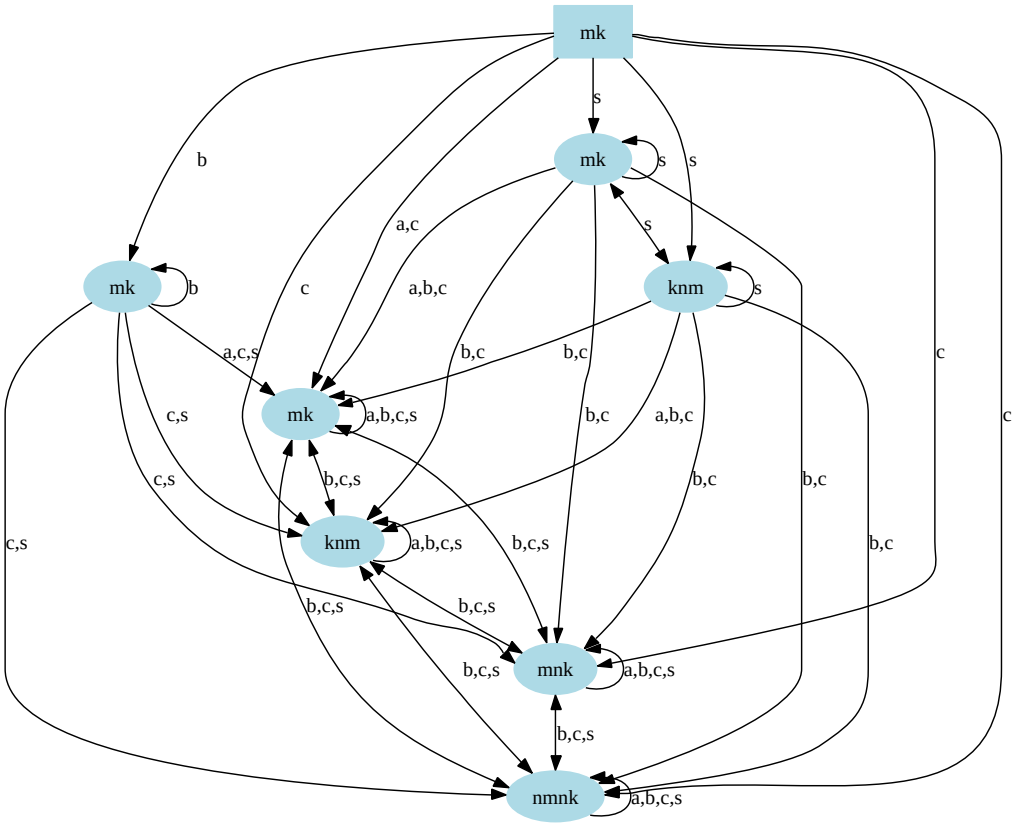

Figure 5.15: The initial state of the protocol.

Initial state of the protocol

First the initial state is taken that is shown in figure 5.12. It shows the initial knowledge of agent a.

agent a believes m

```
["a"]<"K_m">true - TRUE
```

agent b believes m

```
["b"]<"K_m">true - FALSE
```

agent c believes m

```
["c"]<"K_m">true - FALSE
```

agent s believes m

```
["s"]<"K_m">true - FALSE
```

agent a believes k

```
["a"]<"K_k">true - TRUE
```

agent b believes k

```
["b"]<"K_k">true - FALSE
```

agent c believes k

```
["c"]<"K_k">true - FALSE
```

agent s believes k

```
["s"]<"K_k">true - FALSE
```

This shows that only agent a knows the values of message m and key k. All the other agents are ignorant of their values.

After the first step of the protocol

After the first step is modelled with the update tool, verification is done again to find out the knowledge updates for each agents. Agent s has learnt the key k, and this can be verified.

agent a believes m

```
["a"]<"K_m">true - TRUE
```

agent b believes m

["b"]<"K_m">true - FALSE

agent c believes m

["c"]<"K_m">true - FALSE

agent s believes m

["s"]<"K_m">true - FALSE

agent a believes k

["a"]<"K_k">true - TRUE

agent b believes k

["b"]<"K_k">true - FALSE

agent c believes k

["c"]<"K_k">true - FALSE

agent s believes k

["s"]<"K_k">true - TRUE

After the second step of the protocol

After the second step is modelled with the update tool, agent b has learnt the key k, and this can be verified. verification is done again to find out the knowledge updates for each agents.

agent a believes m

["a"]<"K_m">true - TRUE

agent b believes m

["b"]<"K_m">true - FALSE

agent c believes m

["c"]<"K_m">true - FALSE

agent s believes m

["s"]<"K_m">true - FALSE

agent a believes k

["a"]<"K_k">true - TRUE

agent b believes k

["b"]<"K_k">true - TRUE

agent c believes k

```
["c"]<"K_k">true - FALSE
```

agent s believes k

```
["s"]<"K_k">true - TRUE
```

At the end of the protocol

Agent b has learnt the message m at the end of the protocol.

agent a believes m

```
["a"]<"K_m">true - TRUE
```

agent b believes m

```
["b"]<"K_m">true - TRUE
```

agent c believes m

```
["c"]<"K_m">true - FALSE
```

agent s believes m

```
["s"]<"K_m">true - FALSE
```

agent a believes k

```
["a"]<"K_k">true - TRUE
```

agent b believes k

```
["b"]<"K_k">true - TRUE
```

agent c believes k

```
["c"]<"K_k">true - FALSE
```

agent s believes k

```
["s"]<"K_k">true - TRUE
```

This example is taken from the paper on Update Semantics, see [HMV05]. As mentioned before, the models drawn in the paper are done manually by hand. It was found that the models drawn are not completely correct. The pictures shown in this thesis, represent the correct model for each stage of the protocol.

Chapter 6

Conclusion

The concept of Update Semantics is clear now. Update Semantics is a logical language for describing (properties of) runs of security protocols. The semantics of the language is based on traditional Kripke models representing the epistemic state of the agents. Transition rules that precisely indicate the belief updates that happen under certain preconditions, are used to describe the changes in the epistemic state of the agent system as a result of the execution of a protocol. These belief updates give rise to modifications of the models representing the agents' epistemic state in a way that is precisely given by semantic operations on these models. This has been illustrated for three of the well-known security protocols, viz. the SRA Three Pass protocol, the Wide-Mouthed Frog protocol and the Needham Schroeder Key exchange protocol.

This update tool is a milestone in this area of research. Keeping this as the basic building block, research can be expanded in many possible ways. The tool is adaptable and is implemented in a very user friendly manner. The update tool has made update semantics easy for use with large number of agents and/ or more quantity of information. The scalable architecture of the tool in C language makes future extensions simpler. The analysis of Update Semantics can now be done faster and in a more reliable way. The format that is chosen to represent the epistemic states, i.e the AUT format immediately allows visualization and verification. The strengths and weaknesses of Update Semantics can now be better studied by experiments, which hopefully will lead to improvements of the theoretical framework.

There are lot of possibilities for future work. The work mentioned in this thesis, takes one trace of a protocol at a time to model using Update Semantics. As mentioned in the previous chapter, even if the protocol that is modelled is not secure, it cannot be found out by just taking one trace of the protocol. So, generating the traces automatically can be the next step in this domain. This should be an easy matter, since the basic steps are already made automatic. If a software application is developed which produces all the traces for a given

protocol, and then each of these traces can be modelled using the update tools, and eventually the attack can be found out. Another possibility for future work is experimenting this tool with larger protocols, where more agents are involved. The work mentioned in this thesis, is very promising to benefit the security protocol analysis.

Bibliography

[BAN96] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In *Practical Cryptography for Data Internetworks*. 1996.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proceedings CAV'96*, volume 1102 of *LNCS*, pages 437–440, 1996.

[Gra04] 2004. http://www.graphviz.org/.

[HMV05] A. Hommersom, J.-J. Meyer, and E.P. de Vink. Update semantics of security protocols. *Synthese/Knowledge, Rationality and Action*, 2005.

[Hom03] A. Hommersom. Reasoning about security. Master's thesis, Universiteit Utrecht, 2003.

Appendix A

SRA Three Pass Protocol

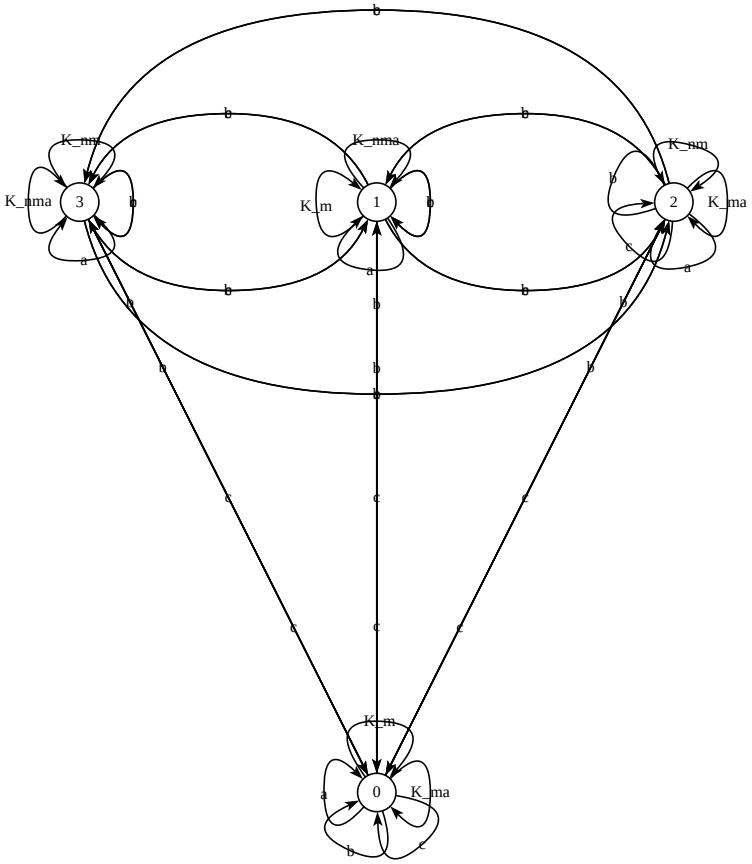

Figure A.1: The initial state represented using the Update Semantics Model

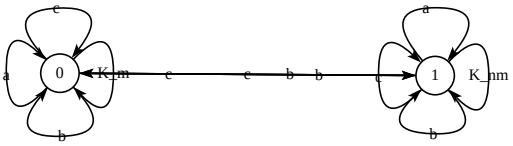

Figure A.2: The Update Semantics Model after doing $Pub(a, m_a)$

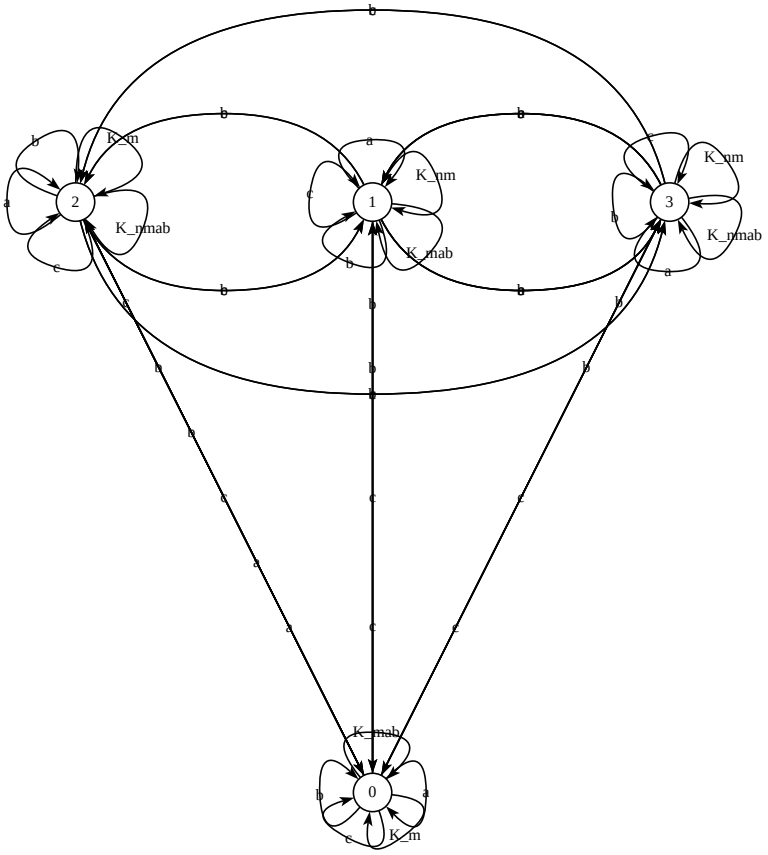

Figure A.3: The Update Semantics Model after doing ADDATOM$_{m_{ab}}$

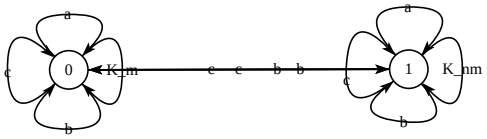

Figure A.4: The Update Semantics Model after doing $Pub(b, m_{ab})$)

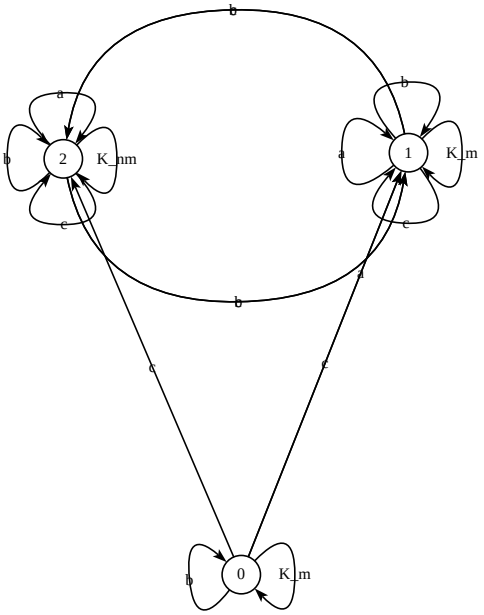

Figure A.5: The Update Semantics Model after doing Update(m_a,b)

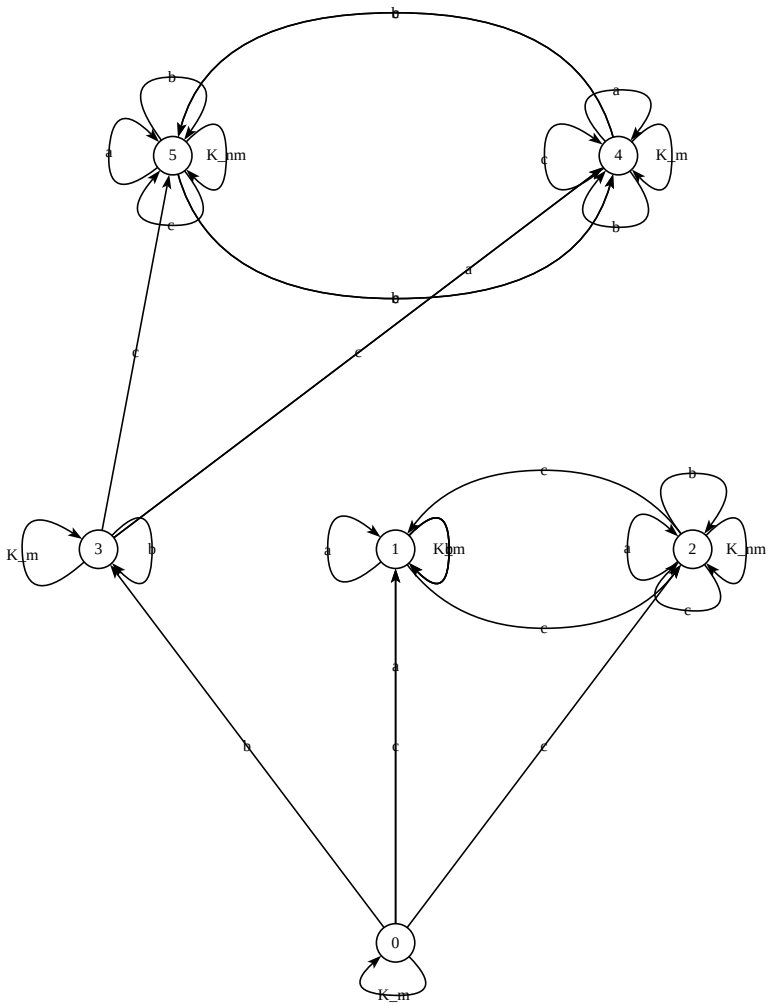

Figure A.6: The final model after doing the side-effect function

Appendix B

Implemented Code

Here in appendix all the implemented code is given with necessary comments. So that any reader can have a quick idea of what is being done in the code.

ussp.C

```
#include <stdio.h>
#include <string.h>
#include "messages.h"
#include "lts.h"
#include "label.h"
#include "AddAtom.h"
#include "Unfold.h"

#define VERSION "Version: 1.1 "
#define IN_FILE   "./in.aut"  /* default input file */
#define OUT_FILE  "./out.aut" /* default output file */

char g_inputFileName[BUFFER_SIZE];
char g_outputFileName[BUFFER_SIZE];
char g_atomName[BUFFER_SIZE];
char g_unfoldAgentsList[BUFFER_SIZE];
char g_updateAgentsList[BUFFER_SIZE];
char g_sideeffectAgentsList[BUFFER_SIZE];
int m_dbg;

int ussp_intro() {

  int iRetVal=0;  /**< number of characters printed */

  iRetVal+=printf("Update semantics of security protocols: %s\n",VERSION);
  iRetVal+=printf("Usage: ussp -i <in> -o <out> [-a <AtomName>] ");
```

```
  iRetVal+=printf("[-f <Agent(s)>] [-u <K,Agent(s)>]
                   [-s <Agent(s),Agent(s),K>] [-d] [-h]\n");
  iRetVal+=printf("in           : input file (.aut) <in>
                                  (default=%s)\n",IN_FILE);
  iRetVal+=printf("out          : output file (.aut) <out>
                                  (default=%s)\n",OUT_FILE);
  iRetVal+=printf("AtomName     : The name of the atom to be added, example:
                                k_n \n");
  iRetVal+=printf("Agent(s)     : Agents to be unfolded, example:
                                \"{(a),(b)}\" OR \"{(a,b),(c)}\" ...\n");
  iRetVal+=printf("K,Agent(s)   : Update one or more agents with knowledge,
                                K \n");
  iRetVal+=printf("Agent(s),Agent(s),K   : sideeffect, example:
                                \"{(a),(b),(K_p)}\" OR \"{(a,b),(c),(K_p)}\" ...\n");
  iRetVal+=printf("             : For example: \"{K_p,b}\" OR \"{K_m,A}\" ...\n");
  iRetVal+=printf("d            : run ussp in debug mode\n");
  iRetVal+=printf("h            : show this manual page\n");
  return(iRetVal);
}

void parse_cmd(int argc, char** argv )
{

    int i;

    // initialize the globals
    strcpy(g_inputFileName, IN_FILE);
    strcpy(g_outputFileName, IN_FILE);
    strcpy(g_atomName, "");
    strcpy(g_unfoldAgentsList, "");
    strcpy(g_sideeffectAgentsList, "");

    m_dbg = 0;

    for (i=1; i<argc; i++)
    {
        if (argv[i][0]!='-')
        {
            printf("missing - in argument %s\n",argv[i]);
        }
        else
        {
                switch (argv[i][1])
            {
                case 'i': strcpy(g_inputFileName, argv[++i]); break;
                case 'o': strcpy(g_outputFileName, argv[++i]); break;
```

```
                case 'a': strcpy(g_atomName, argv[++i]); break;
                case 'f': strcpy(g_unfoldAgentsList, argv[++i]); break;
                case 'u': strcpy(g_updateAgentsList, argv[++i]); break;
                case 's': strcpy(g_sideeffectAgentsList, argv[++i]); break;
                case 'd': m_dbg = atoi(argv[++i]); break;
                    case 'h': ussp_intro(); exit(0);
                    default : printf("can't understand argument %s\n",argv[i]);
                }
            }
    }

}

int main(int argc,char **argv){
    lts_t M;
    lts_t OutputList = NULL;

    parse_cmd(argc, argv);

    if(argc < 2)
    {
        printf( "Insufficient number of arguments\n");
        ussp_intro();
        exit(0);
    }
    /* read the LTS from the AUT file given as first argument */
    M=lts_create();
    lts_read(g_inputFileName,M);
    lts_set_type(M, LTS_BLOCK);
    /* The following is the AddAtom functionality. */
    if(strlen(g_atomName))
    {
        OutputList = lts_AddAtom(g_atomName,M);
    }
    /* The following is the UNFOLD functionality. */
    if(strlen(g_unfoldAgentsList))
    {
        OutputList = lts_Unfold(g_unfoldAgentsList,M);
    }
    /* The following is the UPDATE functionality. */
    if(strlen(g_updateAgentsList))
    {
        OutputList = (lts_t)lts_Update(g_updateAgentsList,M);
    }
    /* The following is the UPDATE functionality. */
    if(strlen(g_sideeffectAgentsList))
```

```
    {
        OutputList = (lts_t)lts_SideEffect(g_sideeffectAgentsList,M);
    }
    if(NULL != OutputList)
    {
        OutputList = lts_DeleteUnreachableStates(OutputList);
        OutputList = (lts_t)lts_NormalizeStates(OutputList, OutputList->root);
        lts_sort(OutputList);
        lts_write(g_outputFileName,OutputList);
        lts_free(OutputList);
    }
    return 0;
}
```

Addatom.C

```
/*
 * Name : AddAtom.c
 * Purpose: This file contains the implementation
 * of all the functionality related to AddAtom.
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/02/20
*/

#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "AddAtom.h"

char NewAtom_buffer[BUFFER_SIZE];
char NotNewAtom_buffer[BUFFER_SIZE];
extern int m_dbg;

lts_t lts_AddAtom(char *pAtomName,lts_t ltsInput)
{
    lts_t ltsOutputList = NULL;
    int i,count;
    int transitions = 0;
    int states = 0;
    char* pNotNewAtom = NULL;

    /* create the list and set its size
     * based on the input list
     */
```

```
if(m_dbg)
printf("lts_AddAtom entered!\n");
ltsOutputList=lts_create();

if (ltsOutputList==NULL)
{
    Fatal(1,1,"out of memory in new_lts");
}
else
{
    if(m_dbg)
    printf("Created list successfully\n");
}
ltsOutputList->states = (ltsInput->states * 2);
lts_set_type(ltsOutputList,LTS_LIST);
if(m_dbg)
printf("Type has been set for list successfully\n");
states = (ltsInput->states * 2);
ltsOutputList->states = states;
transitions = lts_AddAtom_GetNrOfTransitionsForStates(ltsInput);
if(m_dbg)
printf("States = %d >>> Transitions = %d \n", states, transitions);
ltsOutputList->transitions=transitions;
lts_set_size(ltsOutputList,states,transitions);

/*Set the labels */
count=getlabelcount();
// add the new label
if (NULL != pAtomName)
{
    count=getlabelindex(pAtomName,1);
    count=getlabelcount();
    // Get the Not of New atom and add it to the label list
    pNotNewAtom = GetNotOfNewAtom(pAtomName);
    if(NULL != pNotNewAtom)
    {
        strcpy(NewAtom_buffer, pAtomName);
        strcpy(NotNewAtom_buffer, pNotNewAtom);

        count=getlabelindex(pNotNewAtom,1);
        count=getlabelcount();
        if(m_dbg)
        printf("New Not atom is : %s >> %s \n", pAtomName, pNotNewAtom);
    }
}
ltsOutputList->label_string=(char**)malloc(count*sizeof(char*));
```

```
    for(i=0;i<count;i++)
    {
        ltsOutputList->label_string[i]=(char*)getlabelstring(i);
        if(m_dbg)
        printf("Label at pos %d is %s \n", i, getlabelstring(i));
    }
    ltsOutputList->label_count=count;
    lts_AddAtom_CreateNewTransitions(ltsOutputList, ltsInput, pAtomName);
    lts_set_type(ltsOutputList,ltsInput->type);
    return ltsOutputList;
}

char* GetNotOfNewAtom(char* pNewAtom)
{
    char buffer[BUFFER_SIZE];
    char bufferx[BUFFER_SIZE];
    int i = 0, j = 0, count = 0;

    strcpy(buffer, "");
    strcpy(bufferx, "");
    if(NULL != pNewAtom)
    {
        strcpy(bufferx, pNewAtom);
        count = strlen(pNewAtom);

        for( i = 0; i < count; i++)
        {
            if( i > 0 )
            {
                if( '_' == bufferx[i-1])
                {
                    buffer[j++]= 'n';
                }
            }
            buffer[j++]= bufferx[i];
        }
        buffer[j] = '\0';
    }
    if(m_dbg)
    printf("Not of Atom name: Atom= %s NotAtomName = %s \n", pNewAtom, buffer);
    return buffer;
}

int lts_AddAtom_GetNrOfTransitionsForStates(lts_t ltsInput)
{
    int totalCount = 0;
```

```
    int Itransitions = 0;
    int knowledge = 0;
    int i;
    char* pLabel = NULL;

    Itransitions = ltsInput->transitions;
    for(i = 0; i < ltsInput->transitions; i++)
    {
        pLabel = ltsInput->label_string[ltsInput->label[i]];
        if(strncmp(pLabel, "K_", 2 ) == 0 )
        {
            knowledge++;
        }
    }
    Itransitions = Itransitions - knowledge;
    if(m_dbg)
    {
        printf("Number of knowledge Labels are %d \n", knowledge);
        printf("Number of agent transitions are %d \n", Itransitions);
    }
    totalCount = (knowledge * 2) + (Itransitions * 4) +
                                    (ltsInput->states * 2);
    return totalCount;
}

int lts_AddAtom_CreateNewTransitions(lts_t ltsOutputList, lts_t ltsInput,
                                                          char *pAtomName)
{
    int index = 0;
    int src, dest, state;
    char* pLabel = NULL;
    char* Atom = pAtomName;
    char* pNotNewAtom = NULL;
    int i,j, type;
    if (NULL != ltsInput)
    {
        type =  ltsInput->type;
        lts_set_type(ltsInput,LTS_LIST);
        ltsOutputList->root = ltsInput->root;
        state = ltsInput->states;

        // Generate the output list for each item in the input.
        for(i = 0; i < ltsInput->transitions; i++)
        {
            pLabel = ltsInput->label_string[ltsInput->label[i]];
            src = ltsInput->src[i]; dest = ltsInput->dest[i];
```

```
    ltsOutputList->src[index]  = src;
    ltsOutputList->label[index] = ltsInput->label[i];
    ltsOutputList->dest[index]  = dest;
    if(m_dbg)
    {
    printf("TransLoop 1: Src, Dest =  %d, %s, %d for transition %d \n",
        src, pLabel, dest, i);
    }
    index ++;

    /* Check for knowledge transitions */
    if(strncmp(pLabel, "K_", 2 ) == 0 )
    {
        //do nothing
    }
    else
    {
        src = ltsInput->src[i] + state; dest = ltsInput->dest[i];
        ltsOutputList->src[index]  = src;
        ltsOutputList->label[index] = ltsInput->label[i];
        ltsOutputList->dest[index]  = dest;
        index++;
        if(m_dbg)

        src = ltsInput->src[i]; dest = ltsInput->dest[i] + state;
        ltsOutputList->src[index]  = src;
        ltsOutputList->label[index] = ltsInput->label[i];
        ltsOutputList->dest[index]  = dest;
        index++;

    }

    src = ltsInput->src[i] + state; dest = ltsInput->dest[i] + state;
    ltsOutputList->src[index]  = src;
    ltsOutputList->label[index] = ltsInput->label[i];
    ltsOutputList->dest[index]  = dest;
    index ++;

}
//Dealing the Knowledge transitions separately.
for(i = 0; i < ltsInput->states; i++)
{
    if(NULL != pAtomName)
    {
        ltsOutputList->src[index]  = i;
        ltsOutputList->label[index] = getlabelindex(NewAtom_buffer, 0);
```

```
                ltsOutputList->dest[index]  = i;
                index ++;
            }

            if(NULL != pAtomName)
            {
                ltsOutputList->src[index]  = i + state;
                ltsOutputList->label[index] = getlabelindex(NotNewAtom_buffer, 0);
                ltsOutputList->dest[index]  = i + state;
                index ++;
            }
        }
        lts_set_type(ltsInput,type);
    }
    if(m_dbg)
    printf("CreateNewTransitions completed with return value: %d\n", index);
    return index;
}
```

Addatom.h

```
/*
 * Name : AddAtom.h
 * Purpose: All the AddAtom related functionality are listed here.
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/02/20
 *
 *
*/
#ifndef ADDATOM_LIB_H
#define ADDATOM_LIB_H

#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "lts.h"

extern lts_t lts_AddAtom(char *pAtomName,lts_t ltsInput);
extern lts_t lts_Unfold(char *pAgents,lts_t ltsInput);
extern char* GetNotOfNewAtom(char* pNewAtom);
extern int lts_AddAtom_GetNrOfTransitionsPerState(lts_t ltsInput);
extern int lts_AddAtom_CreateNewTransitions(lts_t ltsOutputList,
                                lts_t ltsInput, char *pAtomName);
```

```
#endif
```

Update.C

```
/*
 * Name : Update.c
 * Purpose: This file contains the implementation of UPDATE functionality
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/03/17
 *
*/
#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "Update.h"

char m_UpdateAgent[BUFFER_SIZE];
char m_KnowledgeAgent[BUFFER_SIZE];

/* the 0th item is the state and the 1st item is the index of the knowledge item */
int  m_UpdateStates[BUFFER_SIZE][2];
extern int m_dbg;

int Update_CreateNewTransitions(lts_t ltsOutputList, lts_t ltsInput)
{
    int root, index = 0;
    int src, dest, states;
    char* pLabel = NULL;
    char* pAgent = NULL;
    char* pKnowledge = NULL;
    int i, j, k, type, increment = 1;


    if(m_dbg)
    printf("Started Update_CreateNewTransitions function\n");

    if (NULL != ltsInput)
    {
        type =  ltsInput->type;
        lts_set_type(ltsInput,LTS_LIST);
        ltsOutputList->root = ltsInput->root;
        root = ltsInput->root;
        states = ltsInput->states;
      if(0 != strcmp("A", m_UpdateAgent))
```

```
 {
//New Transitions
// take each transition and check if the agent in each transition corresponds to
// the Update agent given in the input. If it corresponds, then check if the
// knowledge of the destination state is equal to Knowledge agent given in the
// input. if that also corresponds reproduce the transition in the output list.
// Otherwise, do nothing.
// Always consider knowledge transitions separately.

   for(i = 0; i < ltsInput->transitions; i++)
   {
       pLabel = ltsInput->label_string[ltsInput->label[i]];
       src = ltsInput->src[i]; dest = ltsInput->dest[i];
       if(0 == strncmp(pLabel, "K_", 2))
       {
           ltsOutputList->src[index]  = src;
           ltsOutputList->label[index] = ltsInput->label[i];
           ltsOutputList->dest[index]  = dest;
           index++;
       }
       if(0 == strcmp(pLabel, m_UpdateAgent))
       {
           for(k = 0; k < ltsInput->transitions; k++)
           {
               if(-1 != m_UpdateStates[k][0])
               {
                   pKnowledge = NULL;
                   pKnowledge = (char*)getlabelstring(m_UpdateStates[k][1]);
                   if(NULL != pKnowledge)
                   {
                       if( (dest == m_UpdateStates[k][0]) &&
                           (strcmp(pKnowledge, m_KnowledgeAgent ) == 0 ) &&
                           (0 != strncmp(pLabel, "K_", 2)))
                       {
                           ltsOutputList->src[index]  = src;
                           ltsOutputList->label[index] = ltsInput->label[i];
                           ltsOutputList->dest[index]  = dest;
                           index++;
                       }
                   }
               }
           }
       }
// Old Transitions.
// take each transition and reproduce them the same way in the output list.
```

```
// while reproducing the each state should be added with the number of
// input states
// for example (0,a,0) in input list = (3,a,3) in the output list
// if the no. of input states = 3.
      for(i = 0; i < ltsInput->transitions; i++)
      {
          pLabel = ltsInput->label_string[ltsInput->label[i]];
          src = ltsInput->src[i]; dest = ltsInput->dest[i];

          ltsOutputList->src[index]  = src + states;
          ltsOutputList->label[index] = ltsInput->label[i];
          ltsOutputList->dest[index]  = dest + states;
          index++;
      }
// New to Old transitions
// take each transition and check if the agent in each transition corresponds to
// the Update agent given in the input. If it does not correspond, then reproduce
// the transition to the output list, after adding the number of input states to
// the destination state.
// for example if (0,a,0) in input list and a is not the Update agent given in
// the input, = (0,a,3) in the output list, if the no. of input states = 3.
      for(i = 0; i < ltsInput->transitions; i++)
      {
          pLabel = ltsInput->label_string[ltsInput->label[i]];
          src = ltsInput->src[i]; dest = ltsInput->dest[i];
          if(0 != strcmp(pLabel, m_UpdateAgent))
          {
              if(0 != strncmp(pLabel, "K_", 2))
              {
                  ltsOutputList->src[index]  = src;
                  ltsOutputList->label[index] = ltsInput->label[i];
                  ltsOutputList->dest[index]  = dest + states;
                  index++;
              }
          }
      }

  }
  else
  {
      //Always consider knowledge transitions separately.
      for(i = 0; i < ltsInput->transitions; i++)
      {
          pLabel = ltsInput->label_string[ltsInput->label[i]];
          src = ltsInput->src[i]; dest = ltsInput->dest[i];
```

```
            if(0 == strncmp(pLabel, "K_", 2))
            {
                ltsOutputList->src[index]  = src;
                ltsOutputList->label[index] = ltsInput->label[i];
                ltsOutputList->dest[index]  = dest;
                index++;
            }
        }
//Old Transitions.
// take each transition and reproduce them the same way in the output list.
// while reproducing the each state should be added with the number of
// input states
// for example (0,a,0) in input list = (3,a,3) in the output list
// if the no. of input states = 3.

        for(i = 0; i < ltsInput->transitions; i++)
        {
            pLabel = ltsInput->label_string[ltsInput->label[i]];
            src = ltsInput->src[i]; dest = ltsInput->dest[i];


            ltsOutputList->src[index]  = src + states;
            ltsOutputList->label[index] = ltsInput->label[i];
            ltsOutputList->dest[index]  = dest + states;
            index++;
        }
 //New Transitions
 // take all transitions and check if the knowledge of the destination
 // state is equal to Knowledge agent given in the input. if that corresponds
 // then reproduce the transition in the output list.
 // Otherwise, do nothing.

        for(i = 0; i < ltsInput->transitions; i++)
        {
            pLabel = ltsInput->label_string[ltsInput->label[i]];
            src = ltsInput->src[i]; dest = ltsInput->dest[i];
            for(k = 0; k < ltsInput->transitions; k++)
            {
                if(-1 != m_UpdateStates[k][0])
                {
                    pKnowledge = NULL;
                    pKnowledge = (char*)getlabelstring(m_UpdateStates[k][1]);
                    if(NULL != pKnowledge)
                    {
                        if( (dest == m_UpdateStates[k][0]) &&
                           (strcmp(pKnowledge, m_KnowledgeAgent ) == 0 ) &&
```

```
                            (0 != strncmp(pLabel, "K_", 2)))
                        {
                            ltsOutputList->src[index]  = src;
                            ltsOutputList->label[index] = ltsInput->label[i];
                            ltsOutputList->dest[index]  = dest;
                            index++;
                        }
                    }
                }
            }

        }
    }
    }
    return index;
}
int Update_StatesAndKnowledge(lts_t ltsInput)
{
    char know[BUFFER_SIZE];
    int i, j, state = 0;
    char* pLabel = NULL;
    int  states = 0, count = 0;

    state = ltsInput->states;
    lts_set_type(ltsInput,LTS_LIST);

    //  initialize the members
    for(i = 0; i < BUFFER_SIZE; i++)
    {
        for(j = 0; j < 2; j++)
        {
            m_UpdateStates[i][j] = -1;
        }
    }
    // Relate each states with the knowledge attached to it.

    for(i=0; i < ltsInput->transitions; i++)
    {
        pLabel = ltsInput->label_string[ltsInput->label[i]];
        if(strncmp(pLabel, "K_", 2 ) == 0 )
        {
            strcpy(know, pLabel);
            state = ltsInput->src[i];

            m_UpdateStates[i][0] = state;
            m_UpdateStates[i][1] = getlabelindex(know, 0);
```

```
        }
    }
    lts_set_type(ltsInput,LTS_LIST);
    return count;
}

int Update_ExtractAgentsFromInput (lts_t ltsInput, char* pAgents)
{
    char buffer[BUFFER_SIZE];
    char agent[BUFFER_SIZE];
    int open = 0;
    int count;
    int i, index, length = 0;

  //initialize the members
    strcpy(agent, "");
    strcpy(m_UpdateAgent, "");
    strcpy(m_KnowledgeAgent, "");
// Extract the update and knowledge agents from the input parameter pAgents
// and store them in the variables m_UpdateAgent and m_KnowledgeAgent
    if(NULL != pAgents)
    {
        length = strlen(pAgents);
        strcpy(buffer, pAgents);
        buffer[length] = '\0';
        for( i = 0; i < length; i++)
        {
            if( '{' == buffer[i])
            {
                open = 1;
                index = 0;
                i++;
            }
            if( '}' == buffer[i])
            {
                agent[index] = '\0';
                strcpy ( m_UpdateAgent, agent);
                if(m_dbg)
                {
                    printf("labelstring = %s\n", agent);
                }
                open = 0;
            }
            if(1 == open)
            {
                if( ',' == buffer[i])
```

```
                {
                    agent[index] = '\0';
                    if(0 == strncmp(agent, "K_", 2))
                    {
                        strcpy(m_KnowledgeAgent, agent);
                    }
                    else
                    {
                        strcpy(m_UpdateAgent, agent);
                    }
                    if(m_dbg)
                    {
                        printf("labelstring = %s \n", agent);
                    }
                    index = 0;

                }
                else
                {
                    agent[index++] = buffer[i];
                }
            }
        }
    }
    return count;
}

lts_t lts_Update(char *pAgents,lts_t ltsInput)
{
    lts_t ltsOutputList = NULL;
    int i,count;
    int transitions = 0;
    int states = 0;
    /* create the list and set its size
     * based on the input list
     */
    ltsOutputList=lts_create();
    if (ltsOutputList==NULL)
    {
        Fatal(1,1,"out of memory in new_lts");
    }
    else
    {
        if(m_dbg)
        printf("Created list successfully\n");
    }
```

```
    states = ( ltsInput->states * 2);
    lts_set_type(ltsOutputList,LTS_LIST);

    transitions = ( ltsInput->transitions * 4 );
    ltsOutputList->transitions=transitions;
    lts_set_size(ltsOutputList,states,transitions);

    /*Set the labels */
    count=getlabelcount();
    ltsOutputList->label_string=(char**)malloc(count*sizeof(char*));
    for(i=0;i<count;i++)
    {
        ltsOutputList->label_string[i]=(char*)getlabelstring(i);
    }
    ltsOutputList->label_count=count;
    Update_StatesAndKnowledge(ltsInput);
    Update_ExtractAgentsFromInput(ltsInput, pAgents);
    transitions = Update_CreateNewTransitions(ltsOutputList, ltsInput);
    lts_set_size(ltsOutputList,states,transitions);
    lts_set_type(ltsOutputList,LTS_LIST);
    return ltsOutputList;
}
```

Update.h

```
/*
 * Name : Update.c
 * Purpose: This file contains the implementation of UPDATE functionality
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/03/17
 *
 *
*/

#ifndef UPDATE_LIB_H
#define UPDATE_LIB_H
#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "lts.h"

extern int Update_CreateNewTransitions(lts_t ltsOutputList, lts_t ltsInput);
extern int Update_StatesAndKnowledge(lts_t ltsInput);
extern int Update_ExtractAgentsFromInput(lts_t ltsInput, char* pAgents);
```

```
extern lts_t lts_Update(char *pAgents,lts_t ltsInput);
#endif
```

Unfold.C

```
/*
 * Name : Unfold.c
 * Purpose: This file contains the implementation of the UNFOLD functionality.
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/02/20
 *
*/

#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "Unfold.h"

int  m_UnfoldAgents[BUFFER_SIZE][BUFFER_SIZE];
extern int m_dbg;
int Unfold_GetNrOfPartitions(char* pAgents)
{
    int count = 0; // two closing brackets ")" means two partitions
    char buffer[BUFFER_SIZE];
    int i, length = 0;
    if(NULL != pAgents)
    {
        length = strlen(pAgents);
        strcpy(buffer, pAgents);
        buffer[length] = '\0';
        for( i = 0; i < length; i++)
        {
            if( ')' == buffer[i])
            {
                count++;
            }
        }
    }
    return count;
}

int Unfold_GetNrOfTransitionsAndStates(lts_t ltsInput, int partitions,
                                       int *pTransitions, int *pStates){
    int count = 0;
    int transitions = 0;
```

```
    int i, states = 0;
    int type = 0;

    if(NULL != ltsInput)
    {
        type = ltsInput->type;
        lts_set_type(ltsInput,LTS_LIST);

        states = (1 + (ltsInput->states * partitions));
        for(i = 0; i < ltsInput->transitions; i++)
        {
            if(ltsInput->root == ltsInput->src[i])
            {
                count++;
            }
        }
        transitions = (count + (ltsInput->transitions * partitions));
        lts_set_type(ltsInput,type);
    }
    *pTransitions = transitions;
    *pStates = states;
    return count;
}

int Unfold_CreateNewTransitions(lts_t ltsOutputList, lts_t ltsInput,
                                    char* pAgents, int partitions){
    int root, index = 0;
    int src, dest, states;
    char* pLabel = NULL;
    char* pAgent = NULL;
    int i, j, k, type, increment = 1;

    if (NULL != ltsInput)
    {
        type =  ltsInput->type;
        lts_set_type(ltsInput,LTS_LIST);
        ltsOutputList->root = ltsInput->root;
        root = ltsInput->root;
        states = ltsInput->states;

// For every partition, take each agent and check if it matches with the
// label in the input transitions from the root, set the destination
// as destination + increment, where increment is 1 by default and it
// is incremented by the statecount for each partition.
// for every partition
        for(i = 0; i < BUFFER_SIZE; i++)
```

```
        {
            // for every agent in the partition
            for(j = 0; j < BUFFER_SIZE; j++)
            {
                if(-1 != m_UnfoldAgents[i][j])
                {
                    for(k = 0; k < ltsInput->transitions; k++)
                    {
                        pAgent = (char*)getlabelstring(m_UnfoldAgents[i][j]);
                        pLabel = ltsInput->label_string[ltsInput->label[k]];
                        src = ltsInput->src[k]; dest = ltsInput->dest[k];
                        if((root == src) && (0 == strcmp(pLabel, pAgent)))
                        {
                            dest += increment;
                            ltsOutputList->src[index]  = src;
                            ltsOutputList->label[index] = ltsInput->label[k];
                            ltsOutputList->dest[index]  = dest;
                            index++;
                        }
                    }

                }
            }
            increment = increment + states;
        }

 // Generate the output list for each item in the input.
        for(i = 0; i < ltsInput->transitions; i++)
        {
            pLabel = ltsInput->label_string[ltsInput->label[i]];
            src = ltsInput->src[i]; dest = ltsInput->dest[i];
// handle the knowledge transition in the root.
            if((root == src) && (0 == strncmp(pLabel, "K_", 2)))
            {
                ltsOutputList->src[index]  = src;
                ltsOutputList->label[index] = ltsInput->label[i];
                ltsOutputList->dest[index]  = dest;
                index++;
            }
            increment = 1;
            for(j = 0; j < partitions; j++)
            {
                ltsOutputList->src[index]  = src + increment;
                ltsOutputList->label[index] = ltsInput->label[i];
                ltsOutputList->dest[index]  = dest + increment;
                index++;
```

```
                increment = increment + states;
            }
        }
        lts_set_type(ltsInput,type);
    }
    return index;
}
int Unfold_ExtractPartitionsFromInput(lts_t ltsInput, int partitions, char* pAgents)
{
    char buffer[BUFFER_SIZE];
    char agent[BUFFER_SIZE];
    int open = 0, close = 0;
    int i, j, index, length = 0;
    int label = 0, partition = 0, count = 0;

//  initialize the members
    strcpy(agent, "");
    for(i = 0; i < BUFFER_SIZE; i++)
    {
        for(j = 0; j < BUFFER_SIZE; j++)
        {
            m_UnfoldAgents[i][j] = -1;
        }
    }
// Extract the agents from the input parameter pAgent
// and identify the position of the agent in the labels list
// then put the position of label in a 2 dimensional array
// example: m_UnfoldAgents[partition][labelindex]
    if(NULL != pAgents)
    {
        length = strlen(pAgents);
        strcpy(buffer, pAgents);
        buffer[length] = '\0';
        for( i = 0; i < length; i++)
        {
            if( '(' == buffer[i])
            {
                open = 1;
                index = 0;
                i++;
            }
            if( ')' == buffer[i])
            {
                agent[index] = '\0';
                m_UnfoldAgents[partition][label] = getlabelindex(agent, 0);
                open = 0;
```

```
                partition++;
                label = 0;
            }
            if(1 == open)
            {
                if( ',' == buffer[i])
                {
                    agent[index] = '\0';
                    m_UnfoldAgents[partition][label] = getlabelindex(agent, 0);
                    label++;
                    index = 0;
                }
                else
                {
                    agent[index++] = buffer[i];
                }
            }
        }
    }
    return count;
}
lts_t lts_Unfold(char *pAgents,lts_t ltsInput)
{
    int partitions = 0;
    lts_t ltsOutputList = NULL;
    int i,count;
    int transitions = 0;
    int states = 0;
/* create the list and set its size
 * based on the input list
 */
    ltsOutputList=lts_create();
    if (ltsOutputList==NULL)
    {
        Fatal(1,1,"out of memory in new_lts");
    }
    else
    {
        if(m_dbg)
        printf("Created list successfully\n");
    }
    partitions = Unfold_GetNrOfPartitions(pAgents);
    states = ( 1 + (ltsInput->states * partitions));
    lts_set_type(ltsOutputList,LTS_LIST);
    Unfold_GetNrOfTransitionsAndStates(ltsInput, partitions, &transitions, &states);
    ltsOutputList->transitions=transitions;
```

```
    lts_set_size(ltsOutputList,states,transitions);

/*Set the labels */
    count=getlabelcount();
    ltsOutputList->label_string=(char**)malloc(count*sizeof(char*));
    for(i=0;i<count;i++)
    {
        ltsOutputList->label_string[i]=(char*)getlabelstring(i);
    }
    ltsOutputList->label_count=count;
    Unfold_ExtractPartitionsFromInput(ltsInput, partitions, pAgents);
    Unfold_CreateNewTransitions(ltsOutputList, ltsInput, pAgents, partitions);
    lts_set_type(ltsOutputList,LTS_LIST);
    return ltsOutputList;
}
```

Unfold.h

```
/*
 * Name : Unfold.h
 * Purpose: All the Unfold related functionality are listed here.
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/02/20
 *
 *
*/

#ifndef UNFOLD_LIB_H
#define UNFOLD_LIB_H
#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "lts.h"

extern int Unfold_GetNrOfPartitions(char* pAgents);
extern int Unfold_GetNrOfTransitionsAndStates(
        lts_t ltsInput, int partitions, int *pTransitions, int *pStates);
extern int Unfold_CreateNewTransitions(
        lts_t ltsOutputList, lts_t ltsInput, char* pAgents, int partitions);
extern int Unfold_ExtractPartitionsFromInput(
        lts_t ltsInput, int partitions, char* pAgents);
extern lts_t lts_Unfold(char *pAgents,lts_t ltsInput);
#endif
```

Sideeffect.C

```
/*
 * Name : SideEffect.c
 * Purpose: This file contains the implementation of all the SideEffect processess
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/04/20
 *
*/

#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "SideEffect.h"

// 0th array is for the index of labels corresponding to B
// and 1st array is for C when the input is {(B),(C),(K_p)}
// example for B is (a,b,c) and example of C is (e,f)

int  m_SideEffectAgents[2][BUFFER_SIZE];
int  m_AgentsExceptB[BUFFER_SIZE];
char  m_SideEffectKnowledge[BUFFER_SIZE];
char  m_NotSideEffectKnowledge[BUFFER_SIZE];
int m_UnfoldAgents[BUFFER_SIZE][BUFFER_SIZE];
extern int m_dbg;

lts_t SideEffect_AtomSplit(lts_t ltsInput, int states, char* pKnowledge)
{
    int i, j, k, src, dest, labelindex;
    int count, type = 0, index = 0;
    char *pLabel;
    char *pLabel2;
    int discard = 0;
    lts_t ltsOutputList = NULL;
    ltsOutputList=lts_create();
    lts_set_type(ltsOutputList,LTS_LIST);
    lts_set_size(ltsOutputList,ltsInput->states,ltsInput->transitions);
    lts_set_type(ltsInput,LTS_LIST);
    ltsOutputList->root=ltsInput->root;
    /*Set the labels */
    count=getlabelcount();
    ltsOutputList->label_string=(char**)malloc(count*sizeof(char*));
    for(i=0;i<count;i++)
    {
        ltsOutputList->label_string[i]=(char*)getlabelstring(i);
```

```
    }
    ltsOutputList->label_count=count;

    if((NULL != ltsInput) && (NULL != pKnowledge))
    {
        type = ltsInput->type;
        lts_set_type(ltsInput,LTS_LIST);
        states = (1 + states);
        for(i = 0; i < ltsInput->transitions; i++)
        {
            discard = 0;
            src = ltsInput->src[i];
            dest = ltsInput->dest[i];
            labelindex = ltsInput->label[i];
            pLabel = (char *) getlabelstring(ltsInput->label[i]);
            // submodel
            if((src < states) && (dest < states))
            {
                for(j = 0; j < BUFFER_SIZE; j++)
                {
                    // if the label is in C input,
                    if(labelindex == m_SideEffectAgents[1][j])
                    {
                        if( 0 == KnowledgeCheck(src, dest, ltsInput))
                        {
                            discard = 1;
                        }
                    }
                }
            }
            if(discard == 0)
            {
                ltsOutputList->src[index] = src;
                ltsOutputList->dest[index] = dest;
                ltsOutputList->label[index] = ltsInput->label[i];
                index++;
            }
        }
        lts_set_type(ltsInput,type);
    }
    lts_set_size(ltsOutputList,ltsInput->states,index);
    return ltsOutputList;
}

void GetNotOfKnowledge()
{
```

```
    char buffer[BUFFER_SIZE];
    char bufferx[BUFFER_SIZE];
    int i = 0, j = 0, count = 0;

    strcpy(buffer, "");
    strcpy(bufferx, "");
// check if the knowledge is a negative value of knowledge
    count = strlen(m_SideEffectKnowledge);
    if(count > 0)
    {
        strcpy(bufferx, m_SideEffectKnowledge);
        if(strncmp(bufferx, "K_n", 3) == 0)
        {
            for( i = 0; i < count; i++)
            {
                if( i > 0 )
                {
                    if( ( '_' == bufferx[i-1]) &&
                        ( 'n' == bufferx[i]) )
                    {
                        i++;
                    }
                }
                if( i < count )
                {
                    buffer[j++]= bufferx[i];
                }
            }
            buffer[j] = '\0';
        }
        else
        {
            for( i = 0; i < count; i++)
            {
                if( i > 0 )
                {
                    if( '_' == bufferx[i-1])
                    {
                        buffer[j++]= 'n';
                    }
                }
                buffer[j++]= bufferx[i];
            }
            buffer[j] = '\0';
        }
    }
```

```
    strcpy(m_NotSideEffectKnowledge, buffer);
    return;
}
int KnowledgeCheck(int src, int dest, lts_t ltsInput)
{
    int iFound = 0;
    char* srcKnowledge;
    int i, labelindex;
    char* kLabel;
    char knowledge[BUFFER_SIZE];
    char notknowledge[BUFFER_SIZE];

    strcpy(knowledge, "");
    strcpy(notknowledge, "");
    strcpy(knowledge, m_SideEffectKnowledge);
    strcpy(notknowledge, m_NotSideEffectKnowledge);

    for(i = 0; i < ltsInput->transitions; i++)
    {
        kLabel = (char *) getlabelstring(ltsInput->label[i]);

        srcKnowledge = NULL;
        if( (src == ltsInput->src[i]) &&
            (src == ltsInput->dest[i]) )
        {
            if( (0 == strcmp(kLabel, knowledge)) ||
                (0 == strcmp(kLabel, notknowledge)) )
            {
                srcKnowledge = kLabel;
                break;
            }
        }
    }
    if(NULL != srcKnowledge)
    {
        for(i = 0; i < ltsInput->transitions; i++)
        {
            if( (dest == ltsInput->src[i]) &&
                (dest == ltsInput->dest[i]) )
            {
                kLabel = (char *) getlabelstring(ltsInput->label[i]);
                if(0 == strcmp(kLabel, srcKnowledge))
                {
                    iFound = 1;
                }
```

```
            }
        }
    }
    return iFound;
}
void SideEffect_ExtractAgentsExceptB(lts_t ltsInput, char* pAgents)
{
    int found, index, i, j, labelindex, count = 0;
    char* pLabel;
    char  UnfoldAgents1[BUFFER_SIZE];
    char  UnfoldAgents2[BUFFER_SIZE];

    strcpy(UnfoldAgents1, "");
    strcpy(UnfoldAgents2, "");
// for each label index stored in the sideEffectAgents[0]
//(0 is the B items)
    index = 0;
    count=getlabelcount();
    for(i=0;i<count;i++)
    {
        found = 0;
        for(j=0;j<BUFFER_SIZE;j++)
        {
            labelindex = m_SideEffectAgents[0][j];
            if(i == labelindex)
            {
                found = 1;
// for the first iteration, no need for a comma
                if(strlen(UnfoldAgents1) == 0)
                {
                    sprintf(UnfoldAgents1, "{(%s", getlabelstring(i));
                }
                else
                {
                    sprintf(UnfoldAgents1, "%s,%s", UnfoldAgents1, getlabelstring(i));
                }
            }
        }
        if(0 == found)
        {
            pLabel = (char *)getlabelstring(i);
            if(strncmp(pLabel, "K_", 2) == 0)
            {
            }
            else
            {
```

```
                m_AgentsExceptB[index++] = i;
// for the first iteration, no need for a comma
                if(strlen(UnfoldAgents2) == 0)
                {
                    sprintf(UnfoldAgents2, "),(%s", getlabelstring(i));
                }
                else
                {
                    sprintf(UnfoldAgents2, "%s,%s", UnfoldAgents2, getlabelstring(i));
                }
            }
        }
    }
    if((strlen(UnfoldAgents1) > 0 ) && (strlen(UnfoldAgents2) > 0 ))
    {
        sprintf(UnfoldAgents1, "%s%s)}", UnfoldAgents1, UnfoldAgents2);
    }
    else
    {
        strcpy(UnfoldAgents1,"");
    }
    if(NULL != pAgents)
    {
        strcpy(pAgents, UnfoldAgents1);
    }
    return;
}
lts_t SideEffect_ExtractAgentsFromInputAndUnfold(lts_t ltsInput, char* pAgents)
{
    lts_t ltsOutput = NULL;
    char buffer[BUFFER_SIZE];
    char agent[BUFFER_SIZE];
    int open = 0, close = 0;
    int i, j, index, length = 0;
    int label = 0, partition = 0, count = 0;

//  initialize the members
    strcpy(agent, "");
    strcpy(m_SideEffectKnowledge, "");
    for(j = 0; j < BUFFER_SIZE; j++)
    {
        m_SideEffectAgents[0][j] = -1;
        m_SideEffectAgents[1][j] = -1;
    }
// Extract the agents from the input parameter pAgent
// and identify the position of the agent in the labels list
```

```
// then put the position of label in a 2 dimensional array
// example: m_SideEffectAgents[B][labelindex]
    if(NULL != pAgents)
    {
        length = strlen(pAgents);
        strcpy(buffer, pAgents);
        buffer[length] = '\0';
        for( i = 0; i < length; i++)
        {
            if( '(' == buffer[i])
            {
                open = 1;
                index = 0;
                i++;
            }
            if( ')' == buffer[i])
            {
                agent[index] = '\0';
                if(strncmp(agent, "K_", 2) == 0)
                {
                    strcpy(m_SideEffectKnowledge, agent);
                }
                else
                {
                    m_SideEffectAgents[partition][label] = getlabelindex(agent, 0);
                }
                open = 0;
                partition++;
                label = 0;
            }
            if(1 == open)
            {
                if( ',' == buffer[i])
                {
                    agent[index] = '\0';
                    m_SideEffectAgents[partition][label] = getlabelindex(agent, 0);
                    label++;
                    index = 0;

                }
                else
                {
                    agent[index++] = buffer[i];
                }
            }
        }
```

```
        SideEffect_ExtractAgentsExceptB(ltsInput, agent);
        ltsOutput = (lts_t)lts_Unfold(agent, ltsInput);
    }
    return ltsOutput;
}
lts_t lts_SideEffect(char *pAgents,lts_t ltsInput)
{
    int partitions = 0;
    lts_t ltsOutputList1 = NULL;
    int i,count;
    char* pLabel = NULL;
    lts_t ltsOutputList = NULL;
/* create the list and set its size
 * based on the input list
 */
    ltsOutputList1 = SideEffect_ExtractAgentsFromInputAndUnfold(ltsInput, pAgents);
    GetNotOfKnowledge();
    lts_set_type(ltsOutputList1,LTS_LIST);
    ltsOutputList = SideEffect_AtomSplit(
        ltsOutputList1, ltsInput->states, m_SideEffectKnowledge);
    lts_set_type(ltsOutputList,LTS_LIST);
    return ltsOutputList;
}
```

Sideeffect.h

```
/*
 * Name : SideEffect.h
 * Purpose: This file contains all the SideEffect related functionality.
 *
 * History:
 *
 * Author : Sajni Nirmal
 * Date : 2005/04/17
 *
 *
*/
#ifndef SIDEEFFECT_LIB_H
#define SIDEEFFECT_LIB_H
#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#include "lts.h"

extern lts_t lts_SideEffect(char *pAgents,lts_t ltsInput);
#endif
```

lts.C

```
#include <unistd.h>
#include "messages.h"
#include "lts.h"
#include "label.h"
#include <string.h>
#include <malloc.h>

extern int m_dbg;
lts_t lts_create(){
    lts_t lts=(lts_t)malloc(sizeof(struct lts));
    if (lts==NULL) {
        Fatal(1,1,"out of memory in new_lts");
    }
    lts->begin=NULL;
    lts->src=NULL;
    lts->label=NULL;
    lts->dest=NULL;
    lts->type=LTS_LIST;
    lts->transitions=0;
    lts->states=0;
    lts->tau=-1;
    lts->label_string=NULL;
    return lts;
}
void lts_free(lts_t lts){
    realloc(lts->begin,0);
    realloc(lts->src,0);
    realloc(lts->label,0);
    realloc(lts->dest,0);
    free(lts);
}
static void build_block(int states,int transitions,u_int32_t *begin,
                u_int32_t *block,u_int32_t *label,u_int32_t *other){
    int i;
    int loc1,loc2;
    u_int32_t tmp_label1,tmp_label2;
    u_int32_t tmp_other1,tmp_other2;

    for(i=0;i<states;i++) begin[i]=0;
    for(i=0;i<transitions;i++) begin[block[i]]++;
    for(i=1;i<states;i++) begin[i]=begin[i]+begin[i-1];
    for(i=transitions-1;i>=0;i--){
        block[i]=--begin[block[i]];
    }
    begin[states]=transitions;
    for(i=0;i<transitions;i++){
```

```
        if (block[i]==i) {
            continue;
        }
        loc1=block[i];
        tmp_label1=label[i];
        tmp_other1=other[i];
        for(;;){
            if (loc1==i) {
                block[i]=i;
                label[i]=tmp_label1;
                other[i]=tmp_other1;
                break;
            }
            loc2=block[loc1];
            tmp_label2=label[loc1];
            tmp_other2=other[loc1];
            block[loc1]=loc1;
            label[loc1]=tmp_label1;
            other[loc1]=tmp_other1;
            if (loc2==i) {
                block[i]=i;
                label[i]=tmp_label2;
                other[i]=tmp_other2;
                break;
            }
            loc1=block[loc2];
            tmp_label1=label[loc2];
            tmp_other1=other[loc2];
            block[loc2]=loc2;
            label[loc2]=tmp_label2;
            other[loc2]=tmp_other2;
        }
    }
}

void lts_set_type(lts_t lts,LTS_TYPE type){
    int i,j;

    if (lts->type==type) return; /* no type change */

    /* first change to LTS_LIST */
    switch(lts->type){
        case LTS_LIST:
            lts->begin=(u_int32_t*)malloc(sizeof(u_int32_t)*(lts->states+1));
            if (lts->begin==NULL) Fatal(1,1,"out of memory in lts_set_type");
            break;
```

```
            case LTS_BLOCK:
                lts->src=(u_int32_t*)malloc(sizeof(u_int32_t)*(lts->transitions));
                if (lts->src==NULL) Fatal(1,1,"out of memory in lts_set_type");
                for(i=0;i<lts->states;i++){
                    for(j=lts->begin[i];j<lts->begin[i+1];j++){
                        lts->src[j]=i;
                    }
                }
                break;
            case LTS_BLOCK_INV:
                lts->dest=(u_int32_t*)malloc(sizeof(u_int32_t)*(lts->transitions));
                if (lts->dest==NULL) Fatal(1,1,"out of memory in lts_set_type");
                for(i=0;i<lts->states;i++){
                    for(j=lts->begin[i];j<lts->begin[i+1];j++){
                        lts->dest[j]=i;
                    }
                }
                break;
        }
        /* then change to requried type */
        lts->type=type;
        switch(type){
            case LTS_LIST:
                free(lts->begin);
                lts->begin=NULL;
                return;
            case LTS_BLOCK:
                build_block(lts->states,lts->transitions,lts->begin,lts->src,
                                                        lts->label,lts->dest);
                free(lts->src);
                lts->src=NULL;
                return;
            case LTS_BLOCK_INV:
                build_block(lts->states,lts->transitions,lts->begin,lts->dest,
                                                         lts->label,lts->src);
                free(lts->dest);
                lts->dest=NULL;
                return;
        }
    }
    void lts_set_size(lts_t lts,u_int32_t states,u_int32_t transitions){
        lts->states=states;
        lts->transitions=transitions;
        switch(lts->type){
            case LTS_BLOCK:
            case LTS_BLOCK_INV:
```

```
            lts->begin=(u_int32_t*)realloc(lts->begin,sizeof(u_int32_t)*(states+1));
            if (lts->begin==NULL) Fatal(1,1,"out of memory in lts_set_size");
    }
    switch(lts->type){
        case LTS_LIST:
        //case LTS_BLOCK:
        case LTS_BLOCK_INV:
            if(m_dbg)
            printf("inside first switch statement\n");
            lts->src=(u_int32_t*)realloc(lts->src,sizeof(u_int32_t)*transitions);
            if (lts->src==NULL) Fatal(1,1,"out of memory in lts_set_size");
            if(m_dbg)
            printf("going out of switch\n");
    }
    switch(lts->type){
        case LTS_LIST:
        case LTS_BLOCK:
        case LTS_BLOCK_INV:
            lts->label=(u_int32_t*)realloc(lts->label,sizeof(u_int32_t)*transitions);
            if (lts->label==NULL) Fatal(1,1,"out of memory in lts_set_size");
    }
    switch(lts->type){
        case LTS_LIST:
        case LTS_BLOCK:
            lts->dest=(u_int32_t*)realloc(lts->dest,sizeof(u_int32_t)*transitions);
            if (lts->dest==NULL) Fatal(1,1,"out of memory in lts_set_size");
    }
}
void lts_uniq(lts_t lts){
    int i,j,k,count,oldbegin,found;
    lts_set_type(lts,LTS_BLOCK);
    count=0;
    for(i=0;i<lts->states;i++){
        oldbegin=lts->begin[i];
        lts->begin[i]=count;
        for(j=oldbegin;j<lts->begin[i+1];j++){
            found=0;
            for(k=lts->begin[i];k<count;k++){
                if((lts->label[j]==lts->label[k])&&(lts->dest[j]==lts->dest[k])){
                    found=1;
                    break;
                }
            }
            if (!found){
                lts->label[count]=lts->label[j];
                lts->dest[count]=lts->dest[j];
```

```
                count++;
            }
        }
    }
    lts->begin[lts->states]=count;
    lts_set_size(lts,lts->states,count);
}

void lts_sort(lts_t lts)
{
    int i,j,k,l,d;
    lts_set_type(lts,LTS_BLOCK);
    for(i=0;i<lts->states;i++){
        for(j=lts->begin[i];j<lts->begin[i+1];j++){
            l=lts->label[j];
            d=lts->dest[j];
            for(k=j;k>lts->begin[i];k--){
                if (lts->label[k-1]<l) break;
                if ((lts->label[k-1]==l)&&(lts->dest[k-1]<=d)) break;
                lts->label[k]=lts->label[k-1];
                lts->dest[k]=lts->dest[k-1];
            }
            lts->label[k]=l;
            lts->dest[k]=d;
        }
    }
}
/* AUT IO */

static void readaut(FILE *file,lts_t lts){
    int root,transitions,states,t,from_idx,from_end,label_idx,to_idx,i,from,label,to;
    char buffer[BUFFER_SIZE];
    int dest = -1, statecount=0;
    fscanf(file,"des%*[^(]s");
    fscanf(file,"(%d,%d,%d)\n",&root,&transitions,&states);
    lts_set_type(lts,LTS_LIST);
    lts->root=root;
    lts_set_size(lts,states,transitions);
    for(t=0;t<transitions;t++){
        if (fgets(buffer,BUFFER_SIZE,file)==NULL){
            Fatal(1,1,"fgets error (transition %d)",t+1);
        }

        for(i=strlen(buffer);!isdigit(buffer[i]);i--);
        buffer[i+1]=0;
        for(;isdigit(buffer[i]);i--);
```

```
        to_idx=i+1;
        for(;buffer[i]!=',';i--);
        for(i--;isblank(buffer[i]);i--);
        buffer[i+1]=0;

        for(i=0;!isdigit(buffer[i]);i++);
        from_idx=i;
        for(;isdigit(buffer[i]);i++);
        from_end=i;
        for(;buffer[i]!=',';i++);
        for(i++;isblank(buffer[i]);i++);
        buffer[from_end]='\00';
        label_idx=i;

        from=atoi(buffer+from_idx);
        label=getlabelindex(buffer+label_idx,1);
        to=atoi(buffer+to_idx);

        lts->src[t]=from;
        lts->label[t]=label;
        lts->dest[t]=to;

        if(to > dest)
        {
            dest = to;
            statecount++;
        }
    }
    if (states < statecount)
    {
        printf("The number of states found in input file is more than the
                                                states mentioned in the file");
        printf("States mentioned in first line of file = %d >>> States found
                                        in file = %d \n",states, statecount);
        exit(0);
    }
}
/*
 * we map the states of the lts on-the-fly. currently two maps can be chosen:
 *
 * transparent map:
 * #define MAP(s) s
 *
 * forcing root to be state 0:
 * #define MAP(s) ((s==lts->root)?0:((s==0)?lts->root:s))
 */
```

```
#define MAP(s) ((s==lts->root)?0:((s==0)?lts->root:s))

static void writeaut(FILE *file,lts_t lts){
    int i,j;

    fprintf(file,"des(%d,%d,%d)\n",MAP(lts->root),lts->transitions,lts->states);
    switch(lts->type){
    case LTS_LIST:
        for(i=0;i<lts->transitions;i++)
        {
        fprintf(file,"(%d,%s,%d)\n",MAP(lts->src[i]),
            lts->label_string[lts->label[i]],MAP(lts->dest[i]));
        }
        break;
    case LTS_BLOCK:
        for(i=0;i<lts->states;i++){
            for(j=lts->begin[i];j<lts->begin[i+1];j++)
            {
                fprintf(file,"(%d,%s,%d)\n",MAP(i),
                    lts->label_string[lts->label[j]],MAP(lts->dest[j]));
            }
        }
        break;
    case LTS_BLOCK_INV:
        for(i=0;i<lts->states;i++){
            for(j=lts->begin[i];j<lts->begin[i+1];j++){
                fprintf(file,"(%d,%s,%d)\n",MAP(lts->src[j]),
                    lts->label_string[lts->label[j]],MAP(i));
            }
        }
        break;
    default:
        lts_set_type(lts,LTS_LIST);
        writeaut(file,lts);
        break;
    }
}
void DeleteFromArray(int* ppArray, int item)
{
    int i;

    if(NULL != ppArray)
    {
        for(i=0;i<BUFFER_SIZE;i++)
        {
```

```
            if(ppArray[i] == item)
            {
                ppArray[i] = -1;
                break;
            }
        }

    }
    return;
}
void AddToArray(int* ppArray, int item)
{
    int i;
    if(NULL != ppArray)
    {
        for(i=0;i<BUFFER_SIZE;i++)
        {
            if(ppArray[i] == item)
            {
                break;
            }
            else if(ppArray[i] == -1)
            {
                ppArray[i] = item;
                break;
            }
        }
    }
    return;
}

int IsItFoundInArray(int* ppArray, int item)
{
    int iFound = 0;
    int i;

    if(NULL != ppArray)
    {
        for(i=0;i<BUFFER_SIZE;i++)
        {
            if(ppArray[i] == item)
            {
                iFound = 1;
                break;
            }
        }
```

```
    }
    return iFound;
}
int CheckIfArrayIsEmpty(int* ppArray)
{
    int iFound = 0;
    int i;

    if(NULL != ppArray)
    {
        for(i=0;i<BUFFER_SIZE;i++)
        {
            if(ppArray[i] != -1)
            {
                iFound = 1;
                break;
            }
        }
    }
    return iFound;
}
int NoTransitionsFromYesArrayToNoArray(int* ppYesArray, int* ppNoArray, lts_t lts)
{
    int iFound = 0;
    int i;

    lts_set_type(lts,LTS_LIST);

    if((NULL != ppYesArray) && ( NULL != ppNoArray))
    {
        for(i=0; i < lts->transitions; i++)
        {
// 0 = notfound, 1 = found
            if(1 == IsItFoundInArray(ppYesArray, lts->src[i]))
            {
// 0 = notfound, 1 = found
                if(1 == IsItFoundInArray(ppNoArray, lts->dest[i]))
                {
                    AddToArray(ppYesArray, lts->dest[i]);
                    DeleteFromArray(ppNoArray, lts->dest[i]);
                    iFound = 1;
                }
            }
        }
    }
    return iFound;
```

```
}

lts_t lts_NormalizeStates(lts_t lts, int state)
{
    int i;
    int states = 0;
    int found = 0;

    if(NULL == lts)
    {
    }
    else
    {
        states = lts->states;
        if(state < states)
        {
            // first find if the given state is part of the transitions
            for(i=0; i < lts->transitions; i++)
            {
                if( (state == lts->src[i]) || (state == lts->dest[i]) )
                {
                    found = 1;
                    break;
                }
            }
// if the state is not found in the transition,
// decrement all higher states by 1; and continue this function again.
// if the state is found, then look for the next state in the transition list.
            if(0 == found)
            {
                for(i=0; i < lts->transitions; i++)
                {
                    if(state < lts->src[i])
                    {
                        lts->src[i] = (lts->src[i] -1);
                    }
                    if(state < lts->dest[i])
                    {
                        lts->dest[i] = (lts->dest[i] -1);
                    }
                }
            }
            else
            {
                state ++;
            }
```

```
            lts = lts_NormalizeStates(lts, state);
        }
    }
    lts->root=0;
    return lts;
}

lts_t lts_DeleteUnreachableStates(lts_t lts)
{
    int i,j,k,src,dest;
    int count;
    int index = 0;
    int states = 0;
    char* pLabel = NULL;
    lts_t ltsOutputList = NULL;
    int yesarray[BUFFER_SIZE];
    int noarray[BUFFER_SIZE];
    int root;

    for(i=0;i<BUFFER_SIZE;i++)
    {
        yesarray[i] = -1;
        noarray[i] = -1;
    }
    ltsOutputList=lts_create();
    lts_set_type(ltsOutputList,LTS_LIST);
    lts_set_size(ltsOutputList,lts->states,lts->transitions);
    lts_set_type(lts,LTS_LIST);
    lts->root=root;
    if(m_dbg)
/*Set the labels */
    count=getlabelcount();
    ltsOutputList->label_string=(char**)malloc(count*sizeof(char*));
    for(i=0;i<count;i++)
    {
        ltsOutputList->label_string[i]=(char*)getlabelstring(i);
    }
    ltsOutputList->label_count=count;
    for(i=0; i < lts->transitions; i++)
    {
        if(root == lts->src[i])
        {
            AddToArray(yesarray, lts->dest[i]);
        }
    }
    for(i=0; i < lts->transitions; i++)
```

```
    {
        if(0 == IsItFoundInArray(yesarray, lts->dest[i])) // 0 = notfound, 1 = found
        {
            AddToArray(noarray, lts->dest[i]);
        }
    }
// Now, for each item in the yes array check if there is a
// transition to any of the states in the no array.
// If a transition is found, move the state from noarray to yesarray.
    while ( 1 == NoTransitionsFromYesArrayToNoArray(yesarray,noarray,lts))
    {
    }
// If the noarray is empty, the input list is same as the output list.
// otherwise remove the transitions in the input list where the src
// or dest is part of the noarray.
    if(0 == CheckIfArrayIsEmpty(noarray))
    {
        states = lts->states;
        index = lts->transitions;
        for(i=0; i < lts->transitions; i++)
        {
            ltsOutputList->src[i] = lts->src[i];
            ltsOutputList->dest[i] = lts->dest[i];
            ltsOutputList->label[i] = lts->label[i];
        }
    }
    else
    {
        for(i=0; i < lts->transitions; i++)
        {
            src = lts->src[i];
            dest = lts->dest[i];
// 0 = notfound, 1 = found
            if( (1 == IsItFoundInArray(noarray, src)) ||
                 (1 == IsItFoundInArray(noarray, dest)) )
            {

            }
            else
            {
                ltsOutputList->src[index] = src;
                ltsOutputList->dest[index] = dest;
                ltsOutputList->label[index] = lts->label[i];
                index++;
            }
        }
```

```
    }
    lts_free(lts);
    states = 0;
    for(i=0;i<BUFFER_SIZE;i++)
    {
        if(-1 != yesarray[i])
        {
            states++;
        }
    }
    lts_set_size(ltsOutputList,states,index);
    lts_set_type(ltsOutputList,LTS_LIST);
    ltsOutputList->root=0;
    return ltsOutputList;
}

void lts_read(char *name,lts_t lts){
    FILE* file;
    int i,count;

    file=fopen(name,"r");
    if (file == NULL) {
        FatalCall(1,1,"failed to open %s for reading",name);
    }
    readaut(file,lts);
    count=getlabelcount();
    lts->label_string=malloc(count*sizeof(char*));
    for(i=0;i<count;i++){
        lts->label_string[i]=getlabelstring(i);
    }
    if (lts->tau<0) lts->tau=getlabelindex("\"tau\"",0);
    if (lts->tau<0) lts->tau=getlabelindex("tau",0);
    if (lts->tau<0) lts->tau=getlabelindex("i",0);
    lts->label_count=count;
    fclose(file);
}

void lts_write(char *name,lts_t lts){
    FILE* file=fopen(name,"w");
    if (file == NULL) {
        FatalCall(1,1,"could not open %s for output",name);
    }
    writeaut(file,lts);
    fclose(file);
}
```

lts.h

```
#ifndef LTS_H
#define LTS_H

#define LTS_AUTO 0
#define LTS_AUT 1
#define LTS_DIR 3

#include <sys/types.h>
#include "config.h"
#include <stdio.h>
#define BUFFER_SIZE 2048

typedef enum {LTS_LIST,LTS_BLOCK,LTS_BLOCK_INV} LTS_TYPE;

typedef struct lts {
    LTS_TYPE type;
    u_int32_t root;
    u_int32_t transitions;
    u_int32_t states;
    u_int32_t *begin;
    u_int32_t *src;
    u_int32_t *label;
    u_int32_t *dest;
    int tau;
    char **label_string;
    u_int32_t label_count;
} *lts_t;

extern lts_t lts_create();
extern void lts_free(lts_t lts);
extern void lts_set_type(lts_t lts,LTS_TYPE type);
extern void lts_set_size(lts_t lts,u_int32_t states,u_int32_t transitions);
extern void lts_uniq(lts_t lts);
extern void lts_sort(lts_t lts);
extern void lts_read(char *name,lts_t lts);
extern void lts_write(char *name,lts_t lts);
extern lts_t lts_DeleteUnreachableStates(lts_t lts);
#endif
```

label.C

```
/*
 * $Log: label.c,v $
 * Revision 1.1  2002/02/08 12:14:41  sccblom
 * Just saving.
```

```
 *
 */

#include "label.h"
#include "messages.h"
#include <string.h>

#define LABEL_BLOCK 10

static int label_max=0;
static int label_next=0;
static char** labels=(char**)0;

int getlabelindex(char *label, int create){
    int i;
    for(i=0;i<label_next;i++){
        if (!strcmp(label,labels[i])) {
            return i;
        }
    }

    if (!create) return -1;
    if (label_max==label_next) {
        label_max+=LABEL_BLOCK;
        labels=(char**)realloc(labels,label_max*sizeof(char*));
        if (!labels) Fatal(1,1,"Out of memory in getlabelindex");
    }

    labels[label_next]=strdup(label);
    if (!labels[label_next]) Fatal(1,1,"Out of memory in getlabelindex");

    return label_next++;
}
char *getlabelstring(int label){
    return labels[label];
}
int getlabelcount(){
    return label_next;
}
```

label.h

```
/*
 * $Log: label.h,v $
 * Revision 1.1  2002/02/08 12:14:41  sccblom
 * Just saving.
```

```
 *
 */

#ifndef LABEL_H
#define LABEL_H

extern int getlabelindex(char *label, int create);
extern char *getlabelstring(int label);
extern int getlabelcount();
#endif
```

auto-io.C

```
/*
 * $Log: aut-io.c,v $
 * Revision 1.4  2002/05/15 12:21:59  sccblom
 * Added tex subdirectory and MPI prototype.
 *
 * Revision 1.3  2002/02/12 13:33:36  sccblom
 * First test version.
 *
 * Revision 1.2  2002/02/08 17:42:15  sccblom
 * Just saving.
 *
 * Revision 1.1  2002/02/08 12:14:40  sccblom
 * Just saving.
 *
 */

#include "aut-io.h"
#include "label.h"
#include "config.h"
#include "messages.h"
#include <ctype.h>
#include <string.h>

static char *cvs_id="$Id: aut-io.c,v 1.4 2002/05/15 12:21:59 sccblom Exp $";
static char *cvs_id_h=AUT_IO_H;

#define BUFFER_SIZE 2048

void readaut(FILE *file,lts_t lts){
    int root,transitions,states,t,from_idx,from_end,label_idx,to_idx,i,from,label,to;
    char buffer[BUFFER_SIZE];
    fscanf(file,"des%*[^(]s");
    fscanf(file,"(%d,%d,%d)\n",&root,&transitions,&states);
    lts_set_type(lts,LTS_LIST);
```

```
    lts->root=root;
    lts_set_size(lts,states,transitions);
    for(t=0;t<transitions;t++){
        if (fgets(buffer,BUFFER_SIZE,file)==NULL){
            Fatal(1,1,"fgets error (transition %d)",t+1);
        }

        for(i=strlen(buffer);!isdigit(buffer[i]);i--);
        buffer[i+1]=0;
        for(;isdigit(buffer[i]);i--);
        to_idx=i+1;
        for(;buffer[i]!=',';i--);
        for(i--;isblank(buffer[i]);i--);
        buffer[i+1]=0;

        for(i=0;!isdigit(buffer[i]);i++);
        from_idx=i;
        for(;isdigit(buffer[i]);i++);
        from_end=i;
        for(;buffer[i]!=',';i++);
        for(i++;isblank(buffer[i]);i++);
        buffer[from_end]='\00';
        label_idx=i;

        from=atoi(buffer+from_idx);
        label=getlabelindex(buffer+label_idx,1);
        to=atoi(buffer+to_idx);

        lts->src[t]=from;
        lts->label[t]=label;
        lts->dest[t]=to;
    }
}

/*
 * we map the states of the lts on-the-fly. currently two maps can be chosen:
 *
 * transparent map:
 * #define MAP(s) s
 *
 * forcing root to be state 0:
 * #define MAP(s) ((s==lts->root)?0:((s==0)?lts->root:s))
 */

#define MAP(s) ((s==lts->root)?0:((s==0)?lts->root:s))
```

```
void writeaut(FILE *file,lts_t lts){
    int i,j;

    fprintf(file,"des(%d,%d,%d)\n",MAP(lts->root),lts->transitions,lts->states);
    switch(lts->type){
    case LTS_LIST:
        for(i=0;i<lts->transitions;i++){
            fprintf(file,"(%d,%s,%d)\n",MAP(lts->src[i]),
                lts->label_string[lts->label[i]],MAP(lts->dest[i]));
        }
        break;
    case LTS_BLOCK:
        for(i=0;i<lts->states;i++){
            for(j=lts->begin[i];j<lts->begin[i+1];j++){
                fprintf(file,"(%d,%s,%d)\n",MAP(i),
                    lts->label_string[lts->label[j]],MAP(lts->dest[j]));
            }
        }
        break;
    case LTS_BLOCK_INV:
        for(i=0;i<lts->states;i++){
            for(j=lts->begin[i];j<lts->begin[i+1];j++){
                fprintf(file,"(%d,%s,%d)\n",MAP(lts->src[j]),
                    lts->label_string[lts->label[j]],MAP(i));
            }
        }
        break;
    default:
        lts_set_type(lts,LTS_LIST);
        writeaut(file,lts);
        break;
    }
}
```

auto-io.h

```
/*
 * $Log: aut-io.h,v $
 * Revision 1.1  2002/02/08 12:14:40  sccblom
 * Just saving.
 *
 */

#ifndef AUT_IO_H
#define AUT_IO_H "$Id: aut-io.h,v 1.1 2002/02/08 12:14:40 sccblom Exp $"

#include <stdio.h>
```

```
#include "lts.h"

extern void readaut(FILE *file,lts_t lts);

extern void writeaut(FILE *file,lts_t lts);

#endif
```

Messages.C

```
#include <stdarg.h>
#include <stdio.h>
#include "messages.h"

int verbosity;

void Warning(int v,const char *fmt,...) {
    if (v<=verbosity){
        va_list args;
        va_start(args,fmt);
        vfprintf(stderr,fmt,args);
        fprintf(stderr,"\n");
        va_end(args);
    }
}

void WarningCall(int v,const char *fmt,...) {
    if (v<=verbosity){
        va_list args;
        va_start(args,fmt);
        vfprintf(stderr,fmt,args);
        fprintf(stderr,": ");
        perror("");
        va_end(args);
    }
}

void Fatal(int code,int v,const char *fmt,...){
    if (v<=verbosity){
        va_list args;
        va_start(args,fmt);
        vfprintf(stderr,fmt,args);
        fprintf(stderr,"\n");
        va_end(args);
    }
    exit(code);
}
```

```
void FatalCall(int code,int v,const char *fmt,...){
    if (v<=verbosity){
        va_list args;
        va_start(args,fmt);
        vfprintf(stderr,fmt,args);
        fprintf(stderr,": ");
        perror("");
        va_end(args);
    }
    exit(code);
}
```

Messages.h

```
#ifndef MESSAGES_H
#define MESSAGES_H

extern int verbosity;

extern void Warning(int v,const char *fmt,...);
extern void WarningCall(int v,const char *fmt,...);
extern void Fatal(int code,int v,const char *fmt,...);
extern void FatalCall(int code,int v,const char *fmt,...);


#endif
```