Technische Universität München

Fakultät für Informatik

Diplomarbeit

Werkzeugunterstützte Sicherheits-Analyse von kryptographischen Protokollen mit automatischen Theorem-Beweisern

Andreas Gilg

Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Jan Jürjens
Abgabedatum:	15.03.2005

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.03.2005

Andreas Gilg

Inhaltsverzeichnis

Kapitel 1

Einführung

In Zeiten, in denen das Internet immer mehr Verbreitung findet und die Nutzungsmöglichkeiten des Internets rasant zunehmen, nehmen Sicherheitsfragen einen immer größer werdenden Stellenwert ein. Die Datenübertragung, sei es nun über das Internet, ein lokales LAN oder einen beliebigen Netztyp schreitet immer weiter in Bereiche vor, die strikte Sicherheitsvorkehrungen nötig machen. Die Gewährleistung dieser Sicherheit ist aber keineswegs trivial.

Die Kryptographie bietet weitreichende Möglichkeiten Übertragungsprotokolle sicher zu gestalten. Oft wird aber bei der Entwicklung, selbst bei industriell bedeutsamen Projekten, eine Schwachstelle übersehen. Es sollen deshalb Sicherheitsaspekte bereits in der Entwurfsphase überprüft werden können, da Sicherheitsanalysen und Korrekturen von Sicherheitslücken in der Entwurfsphase Zeit und Kosten sparen. Das in dieser Arbeit zu entwickelnde Tool soll dazu beitragen, die Sicherheitsaspekte von sicherheitskritischen Protokollen schon während der Entwicklung zu überprüfen. Die Vorteile einer automatischen Überprüfung liegen darin, dass es nicht nötig ist sich mit Sicherheitsfragen näher auseinanderzusetzen. Es können selbst komplizierte Protokolle einfach durch ungeübte Anwender überprüft werden.

Im Folgenden soll ein Tool entwickelt werden, das Übertragungsprotokolle in Prädikatenlogische Formeln erster Stufe umwandelt und diese als Eingabe für den Theorembeweiser benutzt. Die eigentliche Überprüfung wird in der vorliegenden Arbeit von einem automatischen Theorembeweiser (ATP) für Logik erster Stufe übernommen. Als Theorembeweiser soll der an der Technische Universität entwickelte Beweiser E-SETHEO benutzt werden. Für die Darstellung der Protokolle kommen UML-Sequenzdiagramme zum Einsatz. UML-Sequenzdiagramme haben sich im objektorientierten Entwurf bereits als Standard etabliert und sind somit weit verbreitet und bekannt. Weiter sprechen für UML die Standarderweiterungsmechanismen, welcher sich auch UMLsec bedient um Sicherheitsanforderungen zu formulieren. Außerdem sprechen für UML, dass es als graphische Darstellung eine intuitive Dar-

stellungsmöglichkeit für Übertragungsprotokolle bietet. Des weiteren stehen viele Werkzeuge für die Erstellung und Bearbeitung von UML-Diagrammen zur Verfügung und es besteht eine breite Unterstützung für die Verarbeitung von UML-Diagrammen durch Programmiersprachen, z.B. in Java. Am Lehrstuhl wurde bereits innerhalb der UMLsec-Gruppe ein Framework entwickelt, welches in der Lage ist UML-Diagramme zu verarbeiten. In dieses Framework soll das entwickelte Tool integriert werden um eine einfache Benutzung zu ermöglichen. Das Tool soll dabei vorrangig über das Webinterface des Frameworks bereit gestellt werden.

Im folgenden Kapitel wird auf nötiges Hintergrundwissen eingegangen und eine kurze Übersicht über UML und dabei speziell Sequenzdiagramme gegeben. Außerdem werden die benutzten externen Tools kurz vorgestellt. Danach wird die theoretische Grundlage zur Übersetzung von UML-Sequenzdiagrammen nach TPTP[1], dem Eingabeformat von E-SETHEO erläutert. Im vierten Kapitel wird näher auf einige Implementierungsdetails eingangen. Im darauffolgenden Kapitel wird die Bedienung des Tools erläutert. Dabei wird vor allem auf das Eingabeformat der Diagramme, sowie die Steuerung des Tools eingegangen. Das vorletzte Kapitel behandelt als Fallbeispiel eine Variante des TLS-Protokolls. Dabei wird die Schwachstelle des Protokolls aufgezeigt und eine korrigierte Version des Protokolls auf Sicherheit überprüft. Im letzten Kapitel wird ein Fazit gezogen und weitere Anwendungs- sowie Erweiterungsmöglichkeiten aufgezeigt.

[1] Thousands of Problems for Theorem Provers

Kapitel 2

Hintergrund

2.1 Sicherheit

In allen Lebensbereichen wird Sicherheit in unterschiedlichem Kontext benutzt, selbst in der Informatik wird Sicherheit in verschiedenen Bereichen mit unterschiedlichen Bedeutungen verwendet. In der vorliegenden Arbeit soll Sicherheit gewährleistet werden, indem Angriffe verhindert werden, d.h. man versucht ein System vor Angriffen zu schützen. Es werden dabei Sicherheitsanforderungen definiert, die vom betroffenen System eingehalten werden sollen. Bei der vorliegenden Arbeit werden hauptsächlich Übertragungsprotokolle betrachtet und dabei als Angriffsszenario eine *man-in-the-middle-Attacke*. Man spricht von einer man-in-the-middle-Attacke wenn sich ein Angreifer Zugang zum Datennetz verschafft und alle Nachrichten, die das Datennetz passieren, kontrolliert. Der Angreifer besitzt dabei die Möglichkeit Nachrichten zu lesen, zu löschen und selbst Nachrichten zu verschicken. Das bedeutet aber gleichzeitig, dass alle Protokollteilnehmer nicht sicher sein können wer der Absender einer Nachricht ist, der Angreifer oder der Protokollteilnehmer, der als Absender in der Nachricht angegeben ist. Ein Beispiel für eine man-in-the-middle-Attacke zeigt Bild 2.1. Das Sequenzdiagramm (näheres zu UML-Sequenzdiagrammen in Abschnitt 2.2.1) zeigt einen Client, der versucht mit einem Server zu kommunizieren. Der komplette Datenaustausch läuft dabei aber über den Angreifer, d.h. der Client hat keine direkte Verbindung mit dem Server. Es werden alle Nachrichten der Kommunikationsteilnehmer vom Angreifer entgegengenommen und gegebenenfalls verändert wieder an den anderen Teilnehmer weitergeschickt. So kann im Normalfall nicht davon ausgegangen werden, dass `Nachricht(a)` gleich `Nachricht(a')` ist. Der Server ist aber der Meinung, er kommuniziert mit dem Client und schickt, falls die `Bedingung` wahr ist, eine Antwort in Form von `Rückgabe(b)` an den mutmaßlichen Client zurück. Diese Nachricht wird dann ebenfalls vom Angreifer in eventuell veränderter Form als `Rückgabe(b')` an den Client weitergereicht.

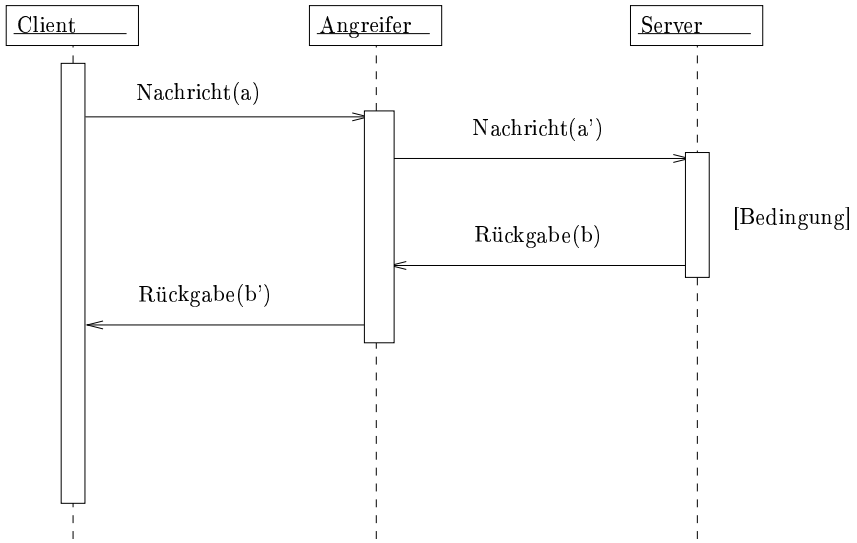

Abbildung 2.1: man-in-the-middle-Attacke

Es wird versucht bei Übertragungsprotokollen durch Kryptographie Sicherheit zu gewährleisten. Man unterscheidet dabei Geheimhaltung, Integrität und Authentizität. Kommen bei der Datenübertragung Verschlüsselungsverfahren zum Einsatz, ist es für den Angreifer nicht so einfach abgefangene Nachrichten zu entschlüsseln. Da der Angreifer aber den kompletten Datenverkehr kontrolliert, kann es vorkommen, dass der Angreifer durch z.B. einen Fehler im Protokoll in Besitz eines Schlüssels kommen kann. Ebenfalls könnte der Angreifer versuchen seinen eigenen Schlüssel in das Protokoll einzuschleusen und ihn als Schlüssel eines Protokollteilnehmers auszugeben.

2.2 UML

Bei der Unified Modeling Language, kurz UML, handelt es sich um eine Modellierungssprache, die Teile der Methoden von Booch, Rumbaugh (OMT; Object Modelling Technique) und Jacobsen vereinheitlicht und erweitert. Als in den frühen 90er Jahren verschiedene objektorientierte Analyse- und Entwurfsmethoden entwickelt wurden, war der Wunsch groß eine einheitliche und standardisierte Methode zu entwerfen. Eine Methode besteht dabei immer aus einer Modellierungssprache und einem Prozess. Der Prozess beschreibt, wie und was bei einem Entwurf durchgeführt werden muss. Die Modellierungssprache dagegen stellt meist den wichtigeren Teil der Entwurfsmethode dar. Die UML wurde im November 1997 von der Object Management Group als Standard akzeptiert. Zur UML gibt es auch einen einheitlichen Prozess, den *Rational Unified Process*. Dieser ist unabhängig von der UML,

d.h. die UML kann als Modellierungssprache auch mit jedem anderen Prozess verwendet werden.

Die UML als grafische Modellierungssprache kann während des gesamten Softwareentwicklungs-Prozesses eingesetzt werden. Hierfür stehen insgesamt neun verschiedene Diagrammarten zur Verfügung. Dies sind im Einzelnen:

- Zustandsdiagramm (engl. statechart diagram)
- Aktivitätsdiagramm (engl. activity diagram)
- Anwendungsfalldiagramm (engl. use-case diagram)
- Klassendiagramm (engl. class diagram)
- Sequenzdiagramm (engl. sequence diagram)
- Kollaborationsdiagramm (engl. collaboration diagram)
- Komponentendiagramm (engl. component diagram)
- Verteilungsdiagramm (engl. deployment diagram)
- Paketdiagramm (engl. package diagram)

2.2.1 Sequenzdiagramme

Sequenzdiagramme und Kollaborationsdiagramme werden unter dem Begriff *Interaktionsdiagramme* zusammengefasst. Interaktionsdiagramme erfassen das Verhalten eines einzelnen Anwendungsfalles und beschreiben, wie einzelne Objekte zusammenarbeiten. Das Hauptaugenmerk liegt dabei auf Nachrichten, die zwischen den einzelnen Objekten ausgetauscht werden.

Kernbestandteil des Sequenzdiagramms sind Objekte, die als Rechtecke dargestellt werden. Jedem Objekt wird dabei eine Spalte im Diagramm zugewiesen, das die Lebenslinie des Objekts als gestrichelte Linie enthält. Der Name des Objekts wird unterstrichen innerhalb des Rechtecks angegeben. Optional kann hinter dem Namen, getrennt durch einen Doppelpunkt, der Klassennamen angegeben werden. Jede Nachricht zwischen zwei Objekten wird als Pfeil zwischen den Lebenslinien der Objekte dargestellt. Die Reihenfolge der Nachrichten ergibt sich dabei aus der Darstellung von oben nach unten. Jede Nachricht wird mindestens mit ihrem Namen gekennzeichnet. Optional können Argumente zwischen Klammern angegeben werden. Bei einem Selbstaufruf handelt es sich um eine Nachricht, die ein Objekt an sich selbst sendet. Dabei wird der Nachrichtenpfeil wieder zurück auf die eigene Lebenslinie gezeichnet. Möchte man sicherstellen, dass die Nachricht nur unter einer bestimmten Bedingung abgeschickt wird, kann eine Einschränkung (engl. guard) in eckigen Klammern angegeben werden. Die Nachricht wird daraufhin nur abgeschickt, wenn die Bedingung wahr ist. Bei komplizierten

Bedingungen sollte zugunsten der Übersichtlichkeit ein eigenes Diagramm gezeichnet werden. Eine weitere Steuermarkierung stellt die Iteration dar. Sie zeigt an, dass eine Nachricht mehrfach auch an unterschiedliche Objekte verschickt wird.

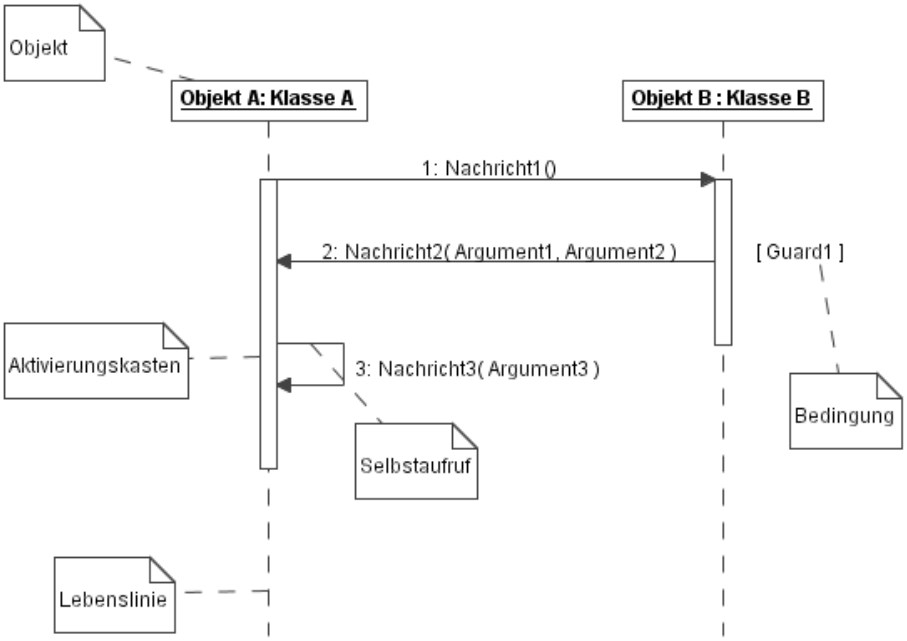

Abbildung 2.2: UML-Sequenzdiagramm Beispiel 1

Bei Nachrichten kann unterschieden werden zwischen synchronen und asynchronen Nachrichten. Synchrone Nachrichten werden mit einer ausgefüllten Pfeilspitze dargestellt. Asynchrone Nachrichten haben den Vorteil, dass der Absender nicht blockiert wird und deshalb seine Ausführung fortsetzen kann. Soll die Rückgabe von Werten explizit angegeben werden, geschieht dies durch die Angabe einer Rückgabe (engl. return), was durch einen Pfeil mit gestrichelter Linie dargestellt wird. Es ist auch möglich mit einer Nachricht ein neues Objekt zu erstellen. Dies wird verdeutlicht indem der Nachrichtenpfeil direkt auf das Objekt zeigt und nicht auf die Lebenslinie. Genauso ist es auch möglich ein Objekt aufzulösen. Dies wird durch ein X unterhalb der Lebenslinie dargestellt. Ein Objekt kann sich selbst auflösen oder durch eine Nachricht gelöscht werden.

Unter einem Aktivierungskasten (engl. activation box) versteht man einen Balken, der über die Lebenslinie gezeichnet wird und darstellt, dass das betreffende Objekt gerade aktiv ist.

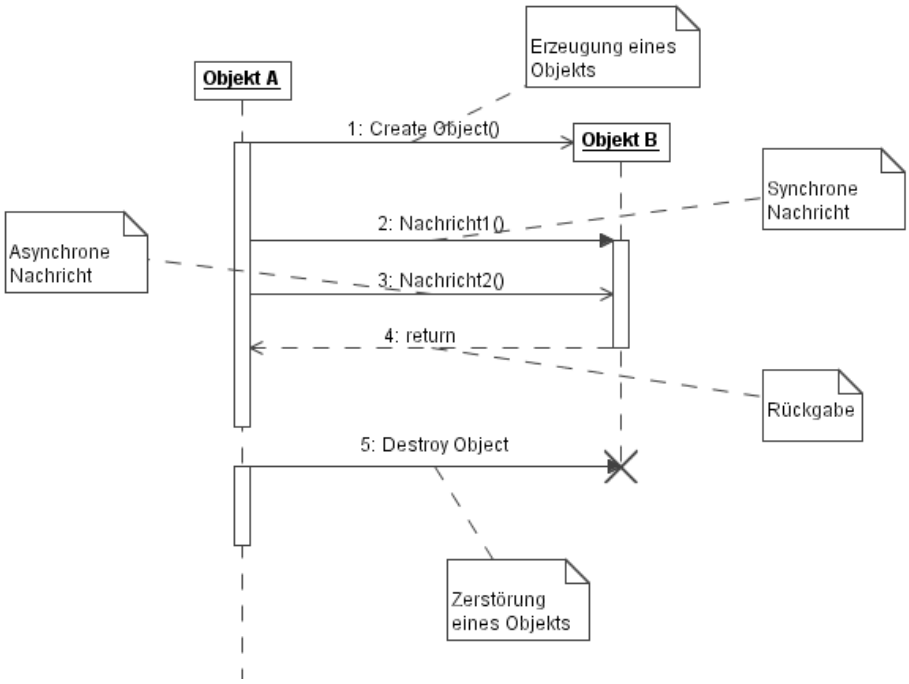

Abbildung 2.3: UML-Sequenzdiagramm Beispiel 2

2.3 Poseidon

Als UML-Editor dient *Poseidon for UML*, welcher aus dem Open-Source-Projekt *ArgoUML* entstanden ist. Poseidon stammt von der Firma Gentleware und steht in mehreren Versionen zur Verfügung. Die Community Edition wird von Gentleware kostenlos bereitgestellt. Natürlich sind dabei einige Einschränkungen in Kauf zu nehmen. So lassen sich in der Community Edition z.B. keine Diagramme drucken. Was aber die Funktionsfähigkeit für unseren Einsatzzweck angeht, sind diese Einschränkungen vernachlässigbar. Poseidon steht im Moment in der Version 3.0.1 zur Verfügung. Für unseren Einsatz wird aber auf die ältere Version 1.6 zurückgegriffen, da für die Versionen ab 2.0 das viki-Framework geändert werden müsste und die neueren Funktionen diesen Aufwand nicht rechtfertigen. Poseidon ist vollständig in Java implementiert und steht deshalb für nahezu alle Plattformen zur Verfügung.

Mit Poseidon wird das Diagramm als so genannte *zargo*-Datei abgespeichert. Eine *zargo*-Datei stellt ein Zip-Archiv dar, das sich mit einem handelsüblichen Packer entpacken lässt. Das Zip-Archiv enthält unter anderem eine XMI-Datei, welche die objektorientierten Modelle in serialisierter Form enthält. XMI[1] dient zur werkzeugunabhängigen Speicherung von Modelldaten

[1] XML Metadata Interchange Format

und wurde von der OMG standardisiert. Wobei die Syntax durch die Metasprache XML[2] und die Semantik durch die UML mit ihrer zugrundliegenden MOF[3] festgelegt wird. Diese von Poseidon erstellte *zargo*-Datei kann direkt ins viki-Framework geladen werden.

2.3.1 Nachteile von Poseidon

Viele Dinge, die sich erst noch in der Standardisierung befinden, werden von Poseidon nicht unterstützt. Meine Beobachtungen beziehen sich dabei vor allem auf Sequenzdiagramme.

Branching Lifelines

Es ist nicht möglich in Poseidon bei Sequenzdiagrammen *branching lifelines* zu zeichnen. Unter *branching lifelines* versteht man Lebenslinien, die sich an einer Stelle unter einer Bedingung teilen, d.h. es besteht die Möglichkeit unterschiedliche Verhalten eines Objekts modellieren zu können.

Bedingungen in Sequenzdiagrammen

Es gibt bis jetzt keine Standardisierung für die Angabe von Bedingungen innerhalb von Sequenzdiagrammen. Poseidon stellt für jede Nachricht nur ein Feld *DispatchAction*, welches den Nachrichtennamen und die Argumente aufnehmen kann, zur Verfügung. Vor Entwicklung des Tools wurde festgelegt, dass Bedingungen in eckigen Klammern vor der eigentlichen Nachricht angegeben werden, wie es eigentlich beim Zeichnen von Sequenzdiagrammen auch gemacht wird. Während der Entwicklung des Tools hat sich dieses Vorgehen allerdings als sehr unübersichtlich beim Zeichnen der Diagramme erwiesen, weshalb alternativ die Bedingungen auch als Tags angegeben werden können. Näheres dazu in Kapitel 5.4.1 auf Seite 43.

Tagged values

Ein nicht nachvollziehbares Problem ist entstanden bei der Benutzung von Tagged Values. Nach mehrmaligem Ändern der Tags wurde festgestellt, dass einzelne Tags beim Löschen zwar aus dem Diagramm verschwinden, aber in der gespeicherten zargo-Datei noch erhalten bleiben und dann im viki-Framework ausgelesen werden. Abhilfe schafft hier nur, wenn beim Löschen eines Tags zuerst der Wert gelöscht wird und dies anschließend mit *Enter* bestätigt wird, d.h. es wird ein Tag mit leerem Wert gespeichert. Anschließend kann das Tag dann gelöscht werden. Es steht dann zwar immer noch in der zargo-Datei, aber ein leeres Tag wird bei der Konvertierung ins TPTP-Format nicht beachtet. Einzige Möglichkeit das Tag wirklich zu entfernen, ist

[2] eXtensible Markup Language
[3] Meta-Object Facility

entweder das Diagramm noch einmal neu zu zeichnen oder die von Poseidon erzeugte XMI-Datei innerhalb der zargo-Datei per Hand zu editieren.

Wiederverwendung von Diagrammen

Wird in Poseidon ein Diagramm als Vorlage für ein neues Diagramm verwendet, müssen unbedingt alle nicht mehr benötigten Objekte gelöscht werden, da es sonst vorkommen kann, dass alte Nachrichten, die von nicht mehr benötigten Objekten ausgehen, nicht gelöscht werden. Diese Nachrichten werden dann bei der Konvertierung ausgelesen und können bei der Erstellung der TPTP-Datei Probleme verursachen.

2.4 E-SETHEO

E-SETHEO wurde am Lehrstuhl entwickelt und ist eine Weiterentwicklung des Theorembeweisers *SETHEO*. Als Eingabesprache benutzt E-SETHEO TPTP. E-SETHEO steht nur in einer Linux-Version zur Verfügung. Die Installation ist etwas schwieriger, da bestimmte Systemvoraussetzungen erfüllt werden müssen. Aber da E-SETHEO am Lehrstuhl entwickelt wurde und dadurch immer eine kompetente Unterstützung zur Verfügung steht, ist im vorliegenden Projekt E-SETHEO als Theorembeweiser erste Wahl. Außerdem ist das vorrangige Ziel, das SequenceAnalyser-Tool über das Webinterface zur Verfügung zu stellen und dabei fällt die eingeschränkte Verfügbarkeit von E-SETHEO nicht ins Gewicht. E-SETHEO steht über die Webseite von Dr. Gernot Stenz (`http://www4.in.tum.de/~stenzg/`) zum Download bereit.

2.5 SPASS

Bei SPASS handelt es sich um einen Theorembeweiser für Logik erster Ordnung mit Gleichheit. Nähere Informationen und Downloads sind auf der SPASS-Webseite [Spa04] zu finden. SPASS steht im Moment in der Version 2.1 sowohl als Webinterface, wie auch für die Plattformen Linux, Solaris und Windows zur Verfügung. Deshalb bietet sich SPASS auch als Theorembeweiser für dieses Projekt an, da es für viele Plattformen verfügbar ist. Allerdings besitzt SPASS eine andere Eingabesyntax als E-SETHEO. Das Eingabeformat von SPASS nennt sich DFG[4], während E-SETHEO TPTP als Eingabe erwartet.

2.6 Tool zur Konvertierung von TPTP nach DFG

Zur Konvertierung der TPTP-Dateien steht ein Tool namens *tptp2X* zur Verfügung. Dieses Tool kann TPTP-Dateien in die verschiedensten Eingabe-

[4] Deutsche Forschungsgemeinschaft

sprachen für Theorembeweiser übersetzen. Es wird innerhalb des SequenceAnalyser-Tools zur Übersetzung vom TPTP-Format ins DFG-Format für den SPASS-Theorembeweiser benutzt. Eine genaue Beschreibung des Tools ist in [SS04] zu finden.

tptp2X kann sowohl direkt unter Linux über die Shell aufgerufen werden als auch innerhalb einer Prolog-Umgebung. Da das Tool für Linux zur Verfügung steht, waren einige Anpassungen nötig um das Tool unter Windows lauffähig zu machen. Da aber nach einer Lösung gesucht wurde um auch unter Windows das SequenceAnalyser-Tool an einen Theorembeweiser anbinden zu können und SPASS als Windows-Programm zur Verfügung steht, war die Anpassung von tptp2X die einzige Möglichkeit. Diese Version des Tools ist nun unter Windows in einer SWI-Prolog-Umgebung lauffähig. Die angepassten tptp2X-Prologdateien stehen über die Webseiten des SequenceAnalyser-Tools [Gil05] zum Download bereit.

2.6.1 Probleme bei der Konvertierung von TPTP nach DFG

Bei der Integration des Tools *tptp2X* traten folgende Probleme zu Tage:

Leerzeichen

Leerzeichen zwischen dem Funktionsnamen und der öffnenden Klammer der Argumente verursachen einen Fehler bei der Konvertierung. Der Programmlauf wird daraufhin mit dem Fehler `operator expected` abgebrochen.

Übersetzung von *true*

tptp2x übersetzt `true` als nullstelliges Prädikat `true_p`, obwohl dies in der DFG-Syntax ebenfalls `true` heißen sollte. Dieser Fehler ist weitaus schwerwiegender, da dadurch SPASS bei allen Protokollen das Ergebnis

```
completion found
```

liefert und dadurch das Protokoll irrtümlich für sicher gehalten wird.

Dieser Fehler kann behoben werden, indem die DFG-Datei nachträglich folgendermaßen bearbeitet wird:

1. Alle Vorkommen von `true_p` durch `true` ersetzen.
2. Im Abschnitt `list_of_symbols` den Ausdruck `(true_p, 0)` löschen.

Von Max Raith existiert ein Skript, welches diese Korrektur bei einer vorliegenden DFG-Datei automatisch durchgeführt. Beim Aufruf von SPASS über das SequenceAnalyser-Tool wird das Problem automatisch behoben, indem nach der Konvertierung von TPTP nach DFG obige Korrektur durchgeführt wird.

Kapitel 3

Konvertierung von Sequenzdiagrammen

Bei der Übersetzung von kryptographischen Protokollen wird versucht das Wissen des Angreifers zu modellieren. Dabei wird von einem man-in-the-middle-Angriff ausgegangen, wie ihn Abbildung 2.1 zeigt. Dies bedeutet, dass der Angreifer alle Nachrichten, die zwischen den Protokollteilnehmern ausgetauscht werden, mitlesen kann. Das Wissen des Angreifers wird so mit jeder verschickten Nachricht erweitert, d.h. das Wissen des Angreifers läßt sich induktiv definieren.

$$\begin{aligned} K^0 &= \text{initiales Wissen des Angreifers vor Protokollausführung} \\ K^{n+1} &= \text{Wissen des Angreifers bis zum } n\text{-ten Schritt} \cup \\ &\quad \text{erlangtes Wissen im } n+1\text{-ten Schritt} \end{aligned}$$

Um das Wissen des Angreifers in Formeln der Logik auszudrücken wird das selbstdefinierte, einstellige Prädikat $knows()$ benutzt. Ein Ausdruck der Form $knows(value)$ bedeutet formal $value \in K^0$ oder $\exists n \in \mathbb{N}.\, (value \in K^n)$. D.h. der Angreifer kannte den Wert $value$ schon vor Protokollausführung oder der Wert ist im Laufe des Protokolls zu seinem Wissen hinzugekommen.

Formal werden bei der Konvertierung von Sequenzdiagrammen Ausdrücke der Prädikatenlogik erster Stufe mit Gleichheit erzeugt, woraus in einem nächsten Schritt TPTP-Ausdrücke erstellt werden. Eine Datei mit allen TPTP-Ausdrücken dient dann als Eingabe für den Theorembeweiser. TPTP-Notation wird aber auch von einer Reihe anderer Theorembeweiser benutzt oder läßt sich mit dem Tool tptp2X in andere Formate übersetzen, wodurch E-SETHEO problemlos durch einen anderen Theorembeweiser ersetzt werden kann.

Innerhalb der TPTP-Datei wird nun versucht das Wissen des Angreifers und die Erweiterung seines Wissens im Laufe des Protokolls zu modellieren. Daraus ergibt sich eine obere Schranke für das Wissen des Angreifers. Eine TPTP-Datei enthält im Groben vier Abschnitte. Dies sind im Einzelnen:

- Axiome oder Regeln für kryptographische Funktionen und selbstdefinierte Funktionen/Prädikate.
- Das Grundwissen des Angreifers vor Protokollausführung
- Die Modellierung des Protokolls
- Der Angriff (engl. Conjecture)

Diese vier Abschnitte werden im Folgenden näher erläutert.

3.1 Axiome

Bei den Axiomen handelt es sich um Regeln, die der Theorembeweiser benutzt, um die Conjecture abzuleiten. Dabei werden für alle Funktionen oder Prädikate, die der Theorembeweiser nicht selbst kennt, Regeln definiert. Diese Regeln werden vom Theorembeweiser zusammen mit der eigentlichen Protokolldefinition verwendet um über die Conjecture eine Aussage treffen zu können. Es sind dabei für alle kryptographischen Funktionen, sowie speziell für das selbstdefinierte Prädikat *knows*() Regeln zu definieren. Es können in einem Protokoll zwar beliebige Funktionen benutzt werden, aber falls für eine Funktion keine Regeln definiert wurden und der Theorembeweiser diese Funktion nicht kennt, so existieren für diese Funktion keine Regeln und der Theorembeweiser kann diese Funktion nicht ableiten. Die vollständigen Axiome sind in Anhang A zu finden. In diesem Abschnitt werden einige Axiome beispielhaft erläutert.

3.1.1 Axiome für das Prädikat *knows*()

Die Eigenschaften des Prädikat *knows*() als logische Formel zeigt Abbildung 3.1.

Aus Formel 3.1 in Abbildung 3.1 geht hervor, welche zweistelligen Funktionen der Angreifer selbst berechnen kann. Dies drückt auch folgender TPTP-Ausdruck aus:

```
input_formula(construct_message_1,axiom,(
! [E1,E2] :
( ( knows(E1)
  & knows(E2) )
=> ( knows(conc(E1, E2))
   & knows(enc(E1, E2))
   & knows(symenc(E1, E2))
   & knows(dec(E1, E2))
   & knows(symdec(E1, E2))
   & knows(ext(E1, E2))
   & knows(sign(E1, E2))
```

$$\begin{aligned}
\forall e1, e2.(knows(e1) \wedge knows(e2) \quad &\Rightarrow \quad knows(e1 :: e2) \\
&\wedge \quad knows(\{e1\}_{e2}) \\
&\wedge \quad knows(\mathtt{Dec}_{e2}(e1)) \\
&\wedge \quad knows(\mathtt{Sign}_{e2}(e1)) \\
&\wedge \quad knows(\mathtt{Ext}_{e2}(e1)) \\
&\wedge \quad knows(\mathtt{Mac}(e1, e2))). \qquad (3.1)
\end{aligned}$$

$$\begin{aligned}
\forall e.(knows(e) \quad &\Rightarrow \quad knows(\mathtt{head}(e)) \\
&\wedge \quad knows(\mathtt{tail}(e)) \\
&\wedge \quad knows(\mathtt{hash}(e))). \qquad (3.2)
\end{aligned}$$

$$\forall e1, e2.(knows(e1 :: e2) \quad \Rightarrow \quad knows(e1) \wedge knows(e2)). \qquad (3.3)$$

$$\forall e, k.(knows(\{e\}_k) \wedge knows(k^{-1}) \quad \Rightarrow \quad knows(e)). \qquad (3.4)$$

$$\forall e, k.(knows(\mathtt{Sign}_k^{-1}(e)) \wedge knows(k) \quad \Rightarrow \quad knows(e)). \qquad (3.5)$$

Abbildung 3.1: Eigenschaften des Prädikats $knows()$

```
   & knows(mac(E1, E2)) ) ) )).
```

Folgender TPTP-Ausdruck ergibt sich aus Formel 3.2 und verdeutlicht welche einstellige Funktionen der Angreifer berechnen kann:

```
input_formula(construct_message_3,axiom,(
! [E] :
( knows(E)
=> ( knows(head(E))
   & knows(tail(E))
   & knows(hash(E)) ) ) )).
```

Außerdem ist der Angreifer in der Lage Listen zu zerlegen und dabei auf einzelne Elemente zuzugreifen. Dies verdeutlicht folgender TPTP-Ausdruck der aus Implikation 3.3 hervorgeht:

```
input_formula(construct_message_2,axiom,(
! [E1,E2] :
( ( knows(conc(E1, E2)) )
=> ( knows(E1)
  & knows(E2) ) ) )).
```

Innerhalb der TPTP-Notation wird bei den Verschlüsselungsverfahren streng zwischen symmetrischer und asymmetrischer Verschlüsselung unterschieden. $enc(e1, e2)$ steht dabei für die asymmetrische Verschlüsselung, während $symenc(e1, e2)$ für die symmetrische Verschlüsselung steht. Das Gleiche gilt auch für die Entschlüsselungsfunktionen ($dec(e1, e2)$ für asymmetrische Entschlüsselung und $symdec(e1, e2)$ für die symmetrische Entschlüsselung). Bei der logischen Formel 3.4 findet diese Unterscheidung nicht statt, deshalb ergeben sich in der TPTP-Notation dafür zwei Formeln:

```
%----------------- Asymmetrical Encryption ------------------

input_formula(enc_equation,axiom,(
! [E1,E2] :
( ( knows(enc(E1, E2))
  & knows(inv(E2)) )
 => knows(E1) ) )).

%----------------- Symmetrical Encryption -------------------

input_formula(symenc_equation,axiom,(
! [E1,E2] :
( ( knows(symenc(E1, E2))
  & knows(E2) )
 => knows(E1) ) )).
```

Aus Implikation 3.5 ergibt sich noch folgender TPTP-Ausdruck:

```
input_formula(sign_equation,axiom,(
! [E,K] :
( ( knows(sign(E, inv(K) ) )
  & knows(K) )
 => knows(E) ) )).
```

3.1.2 Axiome für kryptographische Funktionen

Einige Axiome, die kryptographische Funktionen betreffen, sind bereits in Abschnitt 3.1.1 erläutert worden. Es müssen jetzt nur noch Axiome eingeführt werden, die eine Verbindung zwischen Ver- und Entschlüsselung herstellen.

$$\forall e, k. \quad (\mathtt{Dec}_{k^{-1}}(\{e\}_k) = e).$$
$$\forall e, k. \quad (\mathtt{Ext}_k(\mathtt{Sign}_{k^{-1}}(e) = e).$$

Daraus ergeben sich folgende drei TPTP-Ausdrücke, da hier auch wieder zwischen symmetrischer und asymmetrischer Verschlüsselung unterschieden werden muss:

```
input_formula(dec_axiom,axiom,(
! [E,K] :
( equal( dec(enc(E, K), inv(K)), E ) ) )).

input_formula(symdec_axiom,axiom,(
! [E,K] :
( equal( symdec(symenc(E, K), K), E ) ) )).

input_formula(sign_axiom,axiom,(
! [E,K] :
( equal( ext(sign(E, inv(K)), K), E ) ) )).
```

Im Folgenden werden alle Axiome für die asymmetrische Verschlüsselung exemplarisch näher erläutert.

```
input_formula(dec_axiom,axiom,(
! [E,K] :
( equal( dec(enc(E, K), inv(K)), E ) ) )).
```

Dieses Axiom stellt eine Verknüpfung zwischen der asymmetrischen Verschlüsselung und Entschlüsselung dar. Im Detail bedeutet dies, eine Nachricht E wird mit einem Schlüssel K verschlüsselt. Falls diese Nachricht mit dem inversen Schlüssel von K wieder entschlüsselt wird, erhält man wieder den Klartext der Nachricht E.

```
input_formula(enc_equation,axiom,(
! [E1,E2] :
( ( knows(enc(E1, E2))
  & knows(inv(E2)) )
 => knows(E1) ) )).
```

Dieses Axiom macht deutlich, dass ein Angreifer, der eine verschlüsselte Nachricht $enc(E1, E2)$ abfangen kann (oder eine verschlüsselte Nachricht in seinem Wissen hat) und er den passenden inversen Schüssel $inv(E2)$ dazu besitzt, daraus den Klartext $E1$ der Nachricht bestimmen kann, welcher somit in sein Wissen übergeht und deshalb $knows(E1)$ gilt.

```
input_formula(construct_message_1,axiom,(
! [E1,E2] :
( ( knows(E1)
  & knows(E2) )
=> knows(enc(E1, E2)) ) )).
```

Dieses Axiom bedeutet, dass ein Angreifer die asymmetrische Verschlüsselung $enc(E1, E2)$ berechnen kann, d.h. ist der Angreifer im Besitz eines Wertes und eines Schlüssels, kann er den Klartext $E1$ mit dem bekannten Schlüssel $E2$ verschlüsseln und erweitert sein Wissen um eine verschlüsselte Nachricht $enc(E1, E2)$.

3.1.3 Axiome für Listeneigenschaften

Durch die Konkatenation werden intuitiv Listen definiert. Um innerhalb einer Liste wieder auf einzelne Elemente zugreifen zu können werden die Funktionen $head()$ und $tail()$ benutzt. Die Funktion $head()$ liefert dabei das erste Element einer Liste, also den Listenkopf, $tail()$ dagegen liefert den Rest einer Liste abzüglich des Listenkopfes. Außerdem werden die Funktionen $fst()$, $snd()$, $thd()$ und $frth()$ definiert um leichter auf bestimmte Listenelemente zugreifen zu können. Vor allem bei Bedingungen wird häufig die Gleichheit bestimmter Listenelemente überprüft. Um diese Teile einfacher und übersichtlicher zu halten, wurden diese Funktionen eingeführt.

$$\begin{aligned}
\forall x, y.\quad & (\texttt{head}(x :: y) = x).\\
\forall x, y.\quad & (\texttt{tail}(x :: y) = y).\\
\forall x.\quad & (\texttt{fst}(x) = \texttt{head}(x)).\\
\forall x.\quad & (\texttt{snd}(x) = \texttt{head}(\texttt{tail}(x))).\\
\forall x.\quad & (\texttt{thd}(x) = \texttt{head}(\texttt{tail}(\texttt{tail}(x)))).\\
\forall x.\quad & (\texttt{frth}(x) = \texttt{head}(\texttt{tail}(\texttt{tail}(\texttt{tail}(x))))).
\end{aligned}$$

Übersetzt in TPTP-Notation erhält man folgende Axiome:

```
input_formula(head_axiom,axiom,(
! [X,Y] :
( equal( head(conc(X,Y)), X ) ) )).

input_formula(tail_axiom,axiom,(
! [X,Y] :
( equal( tail(conc(X,Y)), Y ) ) )).

input_formula(fst_axiom,axiom,(
! [X] :
( equal( fst(X), head(X) ) ) )).

input_formula(snd_axiom,axiom,(
! [X] :
( equal( snd(X), head(tail(X)) ) ) )).

input_formula(thd_axiom,axiom,(
! [X] :
( equal( thd(X), head(tail(tail(X))) ) ) )).

input_formula(frth_axiom,axiom,(
! [X] :
( equal( frth(X), head(tail(tail(tail(X)))) ) ) )).
```

3.2 Grundwissen des Angreifers

Das Grundwissen oder initiale Wissen des Angreifers modelliert, mit Hilfe des Prädikats $knows()$, das Wissen des Angreifers vor Protokollausführung. Dies beinhaltet in der Regel die privaten und öffentlichen Schlüssel des Angreifers, sowie eventuell öffentliche Schlüssel der Protokollteilnehmer. Darüber hinaus werden hier noch Werte angegeben, die vor Protokollausführung bereits in das Wissen des Angreifers übergegangen sind.

3.3 Modellierung des Protokolls

Die eigentliche Modellierung des Protokolls orientiert sich am Ansatz, der in [Jür05] beschrieben wird. Jede übertragene Nachricht im Laufe des Protokolls erweitert das Wissen des Angreifers. Auf jede Verknüpfung

$$n = (source(n), guard(n), msg(n), target(n))$$

im Sequenzdiagramm mit der Bedingung $guard(n) = cond(arg_1, \ldots, arg_n)$ und Nachricht $msg(n) = exp(arg_1, \ldots, arg_n)$ wird Formel 3.6 angewandt.

Dabei handelt es sich bei arg_i um Variablen, die während des Protokollablaufs mit Werten gefüllt werden. Es ergibt sich dann für jede Verknüpfung folgende Implikation:

$$\begin{aligned} \forall arg_1, \ldots, arg_n \quad . \quad & (knows(arg_1) \wedge \ldots \wedge knows(arg_n) \\ & \wedge cond(arg_1, \ldots, arg_n) \\ & \Rightarrow knows(exp(arg_1, \ldots, arg_n))) \end{aligned} \tag{3.6}$$

Die Implikation 3.6 modelliert die Erweiterung des Wissens des Angreifers. Das bedeutet, kennt der Angreifer die Werte arg_1 bis arg_n und wird die Bedingung $cond(arg_1, \ldots, arg_n)$ erfüllt, kann der Angreifer die Werte arg_1 bis arg_n dem Protokollteilnehmer schicken und erhält daraufhin die Nachricht $exp(arg_1, \ldots, arg_n)$. Bei diesem Vorgehen werden für jeden Protokollteilnehmer alle abgeschickten Nachrichten betrachtet. Diese Nachrichten werden der Reihe nach durchlaufen und jeweils obige Implikation erstellt. Dadurch wird mit Hilfe aller Nachrichten das Wissen des Angreifers approximiert.

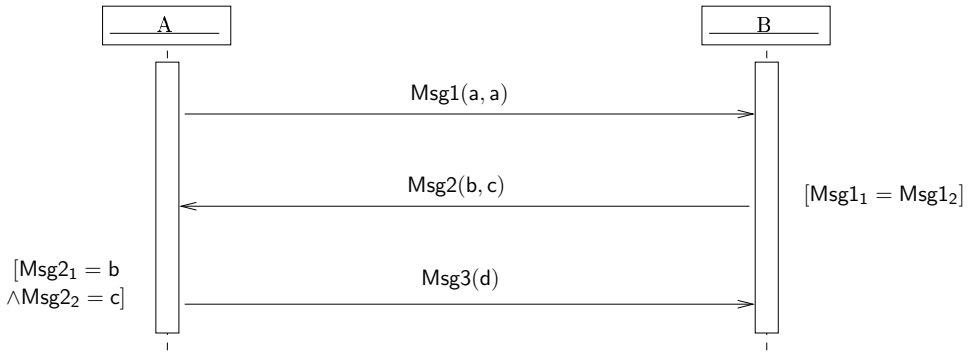

Abbildung 3.2: Beispiel-Protokoll

Im Folgenden wird dieses Vorgehen anhand eines kurzen Beispiel-Protokolls (Abbildung 3.2) näher erläutert. Es wird zuerst Protokollteilnehmer A betrachtet. Er verschickt im Laufe des Protokolls zwei Nachrichten. Zuerst wird mit Nachricht $Msg1(a, a)$ begonnen und darauf Implikation 3.6 angewendet, woraus sich Folgendes ergibt:

$$\begin{aligned} & true && (3.7) \\ \wedge \quad & true && (3.8) \\ \Rightarrow \quad & knows(a) \wedge knows(a) && (3.9) \end{aligned}$$

Da A vor Nachricht $Msg1$ keine Nachricht empfangen hat, ergibt sich an Stelle 3.7 *true*. Außerdem ist keine Bedingung definiert, weshalb sich an Stelle 3.8 ebenfalls *true* ergibt. Innerhalb der Nachricht $Msg1$ wird der Wert a verschickt, d.h. der Wert a wird in das Wissen des Angreifers aufgenommen. Zusammen bedeutet dies: Der Angreifer erhält ohne eine Bedingung erfüllen zu müssen den Wert a.

Nachricht $Msg3(d)$, die zweite Nachricht, die A verschickt und gleichzeitig die dritte Nachricht im Protokollablauf, wird abgeschickt falls die Bedingung $Msg2_1 = b \wedge Msg2_2 = c$ erfüllt ist. Allerdings muss hier beachtet werden, dass A unmittelbar vorher die Nachricht $Msg2(b, c)$ erhalten hat. Dies ergibt dann folgende Implikation:

$$\begin{aligned} & knows(Msg2_1) \wedge knows(Msg2_2) && (3.10)\\ \wedge \quad & (Msg2_1 = b) \wedge (Msg2_2 = c) && (3.11)\\ \Rightarrow \quad & knows(d) && (3.12) \end{aligned}$$

An der Stelle 3.10 werden die Variablen $Msg2_1$ und $Msg2_2$ verwendet und nicht die Werte b und c, da man durch das zugrundeliegende Angreifermodell (Abbildung 2.1) nicht sicher sein kann, dass der Angreifer die Werte b und c auch unverändert an den Teilnehmer A weiterleitet. Es könnten auch beliebige andere Werte sein.

Es wurden nun für den Teilnehmer A zwei Implikationen erstellt. Es ist nun die Frage, ob die Nachrichten im Protokoll streng sequentiell abgearbeitet werden oder ob die Abfolge der Nachrichten beliebig ist. Für den Fall, dass die Nachrichten sequentiell abgeschickt werden, müssen die beide erstellten Implikationen verschachtelt werden. Es ergibt sich dann folgender Term:

$$\begin{aligned} & true \\ & \wedge true \\ & \Rightarrow knows(a) \wedge knows(a) \\ & \quad \wedge \quad (knows(Msg2_1) \wedge knows(Msg2_2) \\ & \qquad \wedge (Msg2_1 = b) \wedge (Msg2_2 = c) \\ & \qquad \Rightarrow knows(d)) \end{aligned} \tag{3.13}$$

Während bei einer beliebigen Abfolge der Nachrichten die oben erstellten Implikationen mit einer Konjunktion verknüpft werden:

$$\begin{aligned} & (true \\ & \wedge true \\ & \Rightarrow knows(a) \wedge knows(a)) \\ \wedge \quad & (knows(Msg2_1) \wedge knows(Msg2_2) \\ & \wedge (Msg2_1 = b) \wedge (Msg2_2 = c) \\ & \Rightarrow knows(d)) \end{aligned}$$

Nachdem alle Nachrichten eines Protokollteilnehmers abgearbeitet sind, wird der nächste Teilnehmer betrachtet. Der Teilnehmer B verschickt im Laufe des Protokolls nur die Nachricht $Msg2(b, c)$. Diese Nachricht wird aber nur abgeschickt, wenn die Bedingung $Msg1_1 = Msg1_2$ erfüllt ist. $Msg1_1$ ist dabei eine Variable, die den Wert des ersten Arguments der Nachricht $Msg1$ enthält. Da man beim vorliegenden Angreifermodell nicht sicher sein kann, dass die Nachricht, die B empfängt, identisch ist zur Nachricht $Msg1(a, a)$, die A abschickt, werden hier nicht die Argumente der Nachricht selbst benutzt, sondern Variablen, die allquantifiziert werden, da der Angreifer jeden beliebigen Wert verschicken kann. Aus Nachricht $Msg2$ ergibt sich dann folgende Implikation:

$$\begin{aligned} & knows(Msg1_1) \wedge knows(Msg1_2) \\ \wedge \quad & (Msg1_1 = Msg1_2) \\ \Rightarrow \quad & knows(b) \wedge knows(c) \end{aligned} \tag{3.14}$$

Da B zusätzlich noch Nachricht $Msg3$ erhält und der Angreifer diese Nachricht ebenfalls abhört, muss dieser Sachverhalt auch noch modelliert werden. Da B danach aber keine Nachricht mehr verschickt, wird anstelle einer Nachricht der Wert $true$ verwendet. Die Modellierung geschieht mit folgender Implikation:

$$\begin{aligned} & knows(Msg3_1) \\ \wedge \quad & true \\ \Rightarrow \quad & true \end{aligned} \tag{3.15}$$

Für Teilnehmer B ergibt sich dann im Gesamten, falls die Nachrichten sequentiell verschickt werden, folgender Ausdruck:

$$\begin{aligned} & knows(Msg1_1) \wedge knows(Msg1_2) \\ & \wedge (Msg1_1 = Msg1_2) \\ & \Rightarrow knows(b) \wedge knows(c) \\ & \quad \wedge \quad (knows(Msg3_1) \\ & \qquad \wedge true \\ & \qquad \Rightarrow true) \end{aligned} \tag{3.16}$$

Geht man nun davon aus, dass die Nachrichten im gesamten Protokoll nur sequentiell abgeschickt werden dürfen, werden alle Ausdrücke, die für die einzelnen Protokollteilnehmer erstellt wurden, über eine Konjunktion verknüpft. Zusätzlich werden alle vorkommenden Variablen all-quantifiziert. Es ergibt sich dann für das Protokoll in Abbildung 3.2 folgende logische Formel:

$$
\begin{array}{ll}
\forall & Msg1_1, Msg1_2, Msg2_1, Msg2_2, Msg3_1. \\
(& (true \\
 & \wedge true \\
 & \Rightarrow knows(a) \wedge knows(a) \\
 & \quad \wedge \quad (knows(Msg2_1) \wedge knows(Msg2_2) \\
 & \qquad \wedge (Msg2_1 = b) \wedge (Msg2_2 = c) \\
 & \qquad \Rightarrow knows(d)\ )\ ) \\
\wedge & (knows(Msg1_1) \wedge knows(Msg1_2) \\
 & \wedge (Msg1_1 = Msg1_2) \\
 & \Rightarrow knows(b) \wedge knows(c) \\
 & \quad \wedge \quad (knows(Msg3_1) \\
 & \qquad \wedge true \\
 & \qquad \Rightarrow true)\ )\ ).
\end{array}
\tag{3.17}
$$

Die logische Formel 3.17 muss nun nur noch in die Eingabesprache des Theorembeweiser, in unserem Fall TPTP, übersetzt werden. Eine Übersicht der TPTP-Notation befindet sich in Tabelle 5.2 und 5.3. Unter Anwendung der TPTP-Funktionen ergibt sich folgender Ausdruck:

```
input_formula(protocol,axiom,(
 ![Msg1_1, Msg1_2, Msg2_1, Msg2_2, Msg3_1] : (
    %A -> Attacker
    (
       ( true
       & true )
      => ( knows(a) & knows(b)
         & ( ( knows(Msg2_1) & knows(Msg2_2)
             & equal(Msg2_1, b) & equal(Msg2_2, c) )
             => knows(d)
           )
         )
     )
 &  %B -> Attacker
     (
       ( knows(Msg1_1) & knows(Msg1_2)
       & equal(Msg1_1, Msg1_2) )
      => ( knows(b) & knows(c)
         & ( ( knows(Msg3_1)
             & true )
             => true
```

```
            )
          )
      )
 )  ) ).
```

3.4 Angriff

Beim Angriff wird eine Hypothese (engl. conjecture) aufgestellt. Diese Conjecture versucht der Theorembeweiser mit Hilfe der Axiome abzuleiten. Der Theorembeweiser betrachtet jedes Modell, das die gegebenen Axiome erfüllt. Das kann aber auch dazu führen, dass Modelle betrachtet werden die zusätzliche Eigenschaften erfüllen, die nicht aus den Axiomen hervorgehen. So kann z.B. ein privater Schlüssel äquivalent zu einem öffentlichen Wert sein den der Angreifer kennt. Dies möchte man aber versuchen zu verhindern und die Sicherheitsüberprüfung unter der Voraussetzung durchführen, dass keine degenerativen Modelle dieser Art auftreten. Dies lässt sich auf folgende Weise verhindern:

1. Es werden zusätzliche Axiome eingeführt die verhindern, dass obiges Problem zu Tage tritt. D.h. die Axiome müssen z.B. gewährleisten, dass ein geheimer Wert verschieden ist zu allen anderen Werten. Dann kann eine Conjecture der Art definiert werden, dass ein Beweis der Conjecture für die Sicherheit des Protokolls steht.

2. Alternativ kann eine negierte Conjecture definiert werden, dann deutet der Beweis der Conjecture auf einen Angriff hin. Falls dagegen kein Beweis für die Conjecture gefunden werden konnte, gilt das Protokoll als sicher. Dies bedeutet, dass alle Modelle, die die vorliegenden Axiome erfüllen, die Conjecture erfüllen müssen um einen Angriff zu ermitteln.

Um sich die Angabe von zusätzlichen Axiomen, wie sie in der erste Option nötig sind, zu ersparen, wird wo immer möglich die zweite Variante verwendet. Es muss dabei beachtet werden, dass falls eine Conjecture nicht abgeleitet werden kann, dies nicht zwingend bedeutet, dass die Negation der Conjecture abgeleitet werden kann.

Um einen Angriff modellieren zu können, muss man sich erst die gestellten Sicherheits-Ziele vor Augen führen. Häufig soll in kryptographischen Protokollen die Geheimhaltung eines Wertes sichergestellt werden. Hierfür wird das selbstdefinierte Prädikat $knows()$ verwendet. Es kommt dabei obige Variante 2 zur Anwendung. Dabei wird überprüft, ob der Angreifer den zu geheimhaltenden Wert kennt. Betrachtet man wieder vorhergehendes Beispiel in Abbildung 3.2 und geht davon aus, dass der Wert d geheimgehalten werden soll, dann wird überprüft ob $knows(d)$ gilt. Es wird dafür in TPTP-Notation folgender Ausdruck benutzt:

```
input_formula(attack,conjecture,(
   knows(d) )).
```

In der vorliegenden Arbeit wird vor allem die Datengeheimhaltung mit dem Prädikat *knows*() überprüft. Ein Beispiel für die Überprüfung von Authentizität anhand eines Purchase-Protokolls ist in [Yua04] zu finden.

Prinzipiell kann in der Conjecture auch jeder andere Ausdruck verwendet werden. Es ist dabei allerdings zu beachten, dass Logik erster Stufe im Allgemeinen nicht berechenbar ist. Es kann deshalb durchaus vorkommen, dass bestimmte Ausdrücke nicht abgeleitet werden können.

3.5 Beispiel für eine Ableitung

In der vorliegenden Arbeit wird der Theorembeweiser als Black-Box betrachtet. Es wird einfach nur die logische Formel erstellt und nach der Überprüfung durch den Theorembeweiser das Ergebnis interpretiert. In diesem Abschnitt soll anhand des vorhergehenden Beispiels aus Abbildung 3.2 versucht werden die Vorgehensweise bei der Ableitung der Conjecture im Ansatz zu verdeutlichen.

In Abschnitt 3.3 wurde für das Diagramm in Abbildung 3.2 der logische Ausdruck 3.17 erstellt. Durch Vereinfachung, indem alle Konstanten *true* weggelassen werden, ergibt sich folgender Ausdruck:

$$
\begin{aligned}
\forall \quad & Msg1_1, Msg1_2, Msg2_1, Msg2_2, Msg3_1. \\
(\quad & (knows(a) \wedge knows(a) \\
& \quad \wedge \quad (knows(Msg2_1) \wedge knows(Msg2_2) \\
& \qquad \wedge (Msg2_1 = b) \wedge (Msg2_2 = c) \\
& \qquad \Rightarrow knows(d) \ ) \ ) \\
\wedge \quad & (knows(Msg1_1) \wedge knows(Msg1_2) \\
& \quad \wedge (Msg1_1 = Msg1_2) \\
& \quad \Rightarrow knows(b) \wedge knows(c) \\
& \qquad \wedge \quad knows(Msg3_1) \ ) \ ).
\end{aligned}
\tag{3.18}
$$

Möchte man nun die Geheimhaltung von d überprüfen, versucht man herauszufinden, ob die Conjecture $knows(d)$ abgeleitet werden kann.

Da bei der vorliegenden Formel alle Variablen all-quantifiziert sind und keine Einschränkung bei der Variablenbelegung gilt, ist die Formel für alle Variablenbelegungen gültig. Man sieht sofort, dass unabhängig von der Variablenbelegung das Prädikat $knows(a)$ gilt. Das bedeutet für das Protokoll, dass ein Angreifer ohne eine Vorbedingung erfüllen zu müssen von Protokollteilnehmer A die Nachricht `Msg1(a,a)` erhält und somit den Wert a in sein Wissen aufnehmen kann.

Möchte man die Implikation in Zeile 3, 4 und 5 der Formel 3.18 für die Ableitung nutzen, muss der Ausdruck

$$knows(Msg2_1) \wedge knows(Msg2_2)$$
$$\wedge(Msg2_1 = b) \wedge (Msg2_2 = c)$$

erfüllt sein. Dies gilt genau dann, wenn $Msg2_1 = b$ und $Msg2_2 = c$. Dies bedeutet für einen Angriff, dass der Angreifer die Nachricht $Msg3$ und damit den Wert d nur erhält, wenn Protokollteilnehmer A die Nachricht $Msg2$ erhalten hat und gleichzeitig die Bedingung $(Msg2_1 = b) \wedge (Msg2_2 = c)$ erfüllt ist. Dies gilt aber nur, wenn der Angreifer die Nachricht $Msg2(b, c)$ an A schickt.

Wir betrachten nun den zweiten Teil der Formel 3.18:

$$\begin{aligned}
&(knows(Msg1_1) \wedge knows(Msg1_2) \\
&\wedge(Msg1_1 = Msg1_2) \\
&\Rightarrow knows(b) \wedge knows(c) \\
&\quad \wedge \quad knows(Msg3_1)\ )
\end{aligned} \tag{3.19}$$

Um diese Implikation ebenfalls für die Ableitung benutzen zu können, müssen die Variablen $Msg1_1$ und $Msg1_2$ mit dem selben Wert belegt werden. Für den Angreifer heißt dies, er erhält von B nur eine Nachricht mit den Werten b und c, wenn B vorher eine Nachricht mit zwei Argumenten, die den gleichen Wert enthalten, empfängt. Wir betrachten hier bespielhaft die Belegung $Msg1_1 = Msg1_2 = a$.

Wird auch noch Variable $Msg3_1$ mit einem Wert belegt, ergibt sich folgende Variablenbelegung:

$$\begin{aligned}
Msg1_1 &= a \\
Msg1_2 &= a \\
Msg2_1 &= b \\
Msg2_2 &= c \\
Msg3_1 &= d
\end{aligned}$$

Eingesetzt in Formel 3.18 ergibt sich folgender Ausdruck:

$$
\begin{array}{ll}
(& (knows(a) \wedge knows(a) \\
& \quad \wedge \quad (knows(b) \wedge knows(c) \\
& \qquad \wedge (b = b) \wedge (c = c) \\
& \qquad \Rightarrow knows(d)\ )\ ) \\
\wedge & (knows(a) \wedge knows(a) \\
& \wedge (a = a) \\
& \Rightarrow knows(b) \wedge knows(c) \\
& \quad \wedge \quad knows(d)\ )\ ).
\end{array} \tag{3.20}
$$

Diese Formel kann noch vereinfacht werden:

$$
\begin{array}{ll}
(& (knows(a) \\
& \quad \wedge \quad (knows(b) \wedge knows(c) \\
& \qquad \Rightarrow knows(d)\ )\ ) \\
\wedge & (knows(a) \\
& \Rightarrow knows(b) \wedge knows(c) \wedge knows(d)\ )\ ).
\end{array} \tag{3.21}
$$

Betrachtet man von Formel 3.21 nur den zweiten Teil

$$
\begin{array}{l}
knows(a) \\
\Rightarrow knows(b) \wedge knows(c) \wedge knows(d)
\end{array} \tag{3.22}
$$

bedeutet dies wörtlich, falls der Angreifer den Wert a kennt, kann er in Besitz der Werte b, c und d kommen. Für die Conjecture $knows(d)$ heißt dies, falls $knows(a)$ erfüllt ist, gilt auch $knows(d)$. Da wir aber im vorliegenden Fall von einer man-in-the-middle-Attacke ausgehen, d.h. der Angreifer hört alle verschickten Nachrichten ab, gilt $knows(a) = true$ nachdem der Angreifer die erste Nachricht, in der a verschickt wurde, abgehört hat. Betrachtet man den weiteren Protokollablauf, so kann der Angreifer eine Nachricht $Msg1(a, a)$ an B schicken und erhält von B die Nachricht $Msg2(b, c)$. Schickt der Angreifer diese Nachricht weiter an A, so erhält er von A die Nachricht $Msg3(d)$ mit dem zu geheimhaltenden Wert d. Der Angreifer erlangt somit Wissen über den geheimen Wert d.

Wäre die Variable $Msg3_1$ mit einem anderen Wert belegt, müsste die Argumentation etwas anders lauten. Gilt z.B. $Msg3_1 = e$ und alle anderen

Belegungen wie oben, erhält man folgende Formel:

$$\begin{aligned}
&(\quad (knows(a) \\
&\qquad \wedge \quad (knows(b) \wedge knows(c) \\
&\qquad\qquad \Rightarrow knows(d)\ )\ ) \\
&\wedge \quad (knows(a) \\
&\qquad \Rightarrow knows(b) \wedge knows(c) \wedge knows(e)\ )\ ).
\end{aligned} \tag{3.23}$$

Geht man ebenfalls davon aus, dass $knows(a) = true$ gilt, so sind laut Implikation in Zeile 4 und 5 die Prädikate $knows(b)$, $knows(c)$ und $knows(e)$ erfüllt. Weiß man nun, dass $knows(b)$ und $knows(c)$ erfüllt sind, kann man dies für die Implikation in Zeile 2 und 3 benutzen und folgern, dass auch $knows(d)$ gilt.

Die Formeln, die durch den Ansatz in Abschnitt 3.3 erstellt werden, gelten zunächst einmal für alle Variablenbelegungen. Möchte man aber die Implikationen innerhalb der Formeln für die Ableitung nutzen, müssen Variablenbelegungen gewählt werden, die die Vorbedingungen der Implikationen erfüllen. Da diese Vorbedingungen häufig $knows()$-Prädikate enthalten und um diese Prädikate zu erfüllen, testet man oft Werte, die sich im Grundwissen des Angreifers befinden oder die der Angreifer bereits während des Protokollablaufs empfangen konnte. Insbesonders sind auch Angriffe denkbar, in denen der Angreifer teilweise Nachrichten unverändert weiterschickt. Oft kommen auch Angriffe vor, in denen der Angreifer die Schlüssel der Protokollteilnehmer durch seinen eigenen Schlüssel ersetzt.

Mit diesem Vorgehen wurde für dieses Beispiel festgestellt, dass ein Angriff existiert und gleichzeitig wurde eine konkrete Variablenbelegung für den Angriff gefunden. Dies ist beim Einsatz eines automatischen Theorembeweisers nicht möglich, sondern es kann nur die Möglichkeit eines Angriffs bewiesen werden, aber nicht eine konkrete Variablenbelegung. Um eine konkrete Variablenbelegung zu erhalten, kann ein ähnlicher Ansatz in Prolog (siehe [Kop05]) benutzt werden. Dieser Ansatz ist aber nicht annähernd so effizient wie die Benutzung eines Theorembeweisers.

Kapitel 4

Implementierung

In diesem Kapitel wird auf einige Implementierungsdetails eingegangen. Es wird dabei bewusst davon Abstand genommen den Quelltext zu erläutern. Dieser steht allen Interessierten zum Herunterladen über die Download-Seite [Vik05] des viki-Frameworks zur Verfügung. Der Quelltext ist im komprimierten viki-Framework enthalten. Es finden sich im Quelltext ausreichend Kommentare um die Funktionsweise verstehen zu können. Im Folgenden wird der grundlegende Algorithmus zur Erstellung der TPTP-Datei erläutert, sowie auf einige Besonderheiten und Probleme näher eingegangen. Außerdem wird die inkrementelle Entwicklung des Tools anhand der immer besser gewordenen Darstellungsmöglichkeit der Diagramme erläutert.

4.1 Algorithmus für die Erstellung der TPTP-Datei

Eine zentrale Funktion im SequenceAnalyser-Tool ist die Erstellung der TPTP-Datei. Die TPTP-Datei wird in Form eines Strings von der Funktion `getTptp` erzeugt. Diese Funktion erwartet als Parameter ein Array mit den Nachrichten in Form von `SeqMsg`-Objekten und den vom Framework gelieferten `IMdrContainer` mit dem UML-Diagramm. Diese Funktion kann auch von anderen Tools benutzt werden, falls diese die TPTP-Datei zur weiteren Bearbeitung benötigen. In diesem Abschnitt wird das Vorgehen der Funktion `getTptp` in groben Zügen erläutert.

Der Algorithmus orientiert sich an der Vorgehensweise, die in Kapitel 3 beschrieben ist. Die beiden Aktivitätsdiagramme in Abbildung 4.1 und 4.2 erläutern das genaue Vorgehen.

Wird über das viki-Framework das Kommando zur Anzeige der TPTP-Datei aufgerufen, werden zuerst alle Nachrichten aus dem MDR-Container, der vom Framework geliefert wird, ausgelesen. Um die Nachrichten verwalten zu können, wurde eine spezielle Klasse `SeqMsg` definiert. Diese Klasse enthält auch alle Methoden, die zur Bearbeitung der einzelnen Nachrichten dienen. Für jede Nachricht wird ein eigenes Objekt der Klasse `SeqMsg` instanziert

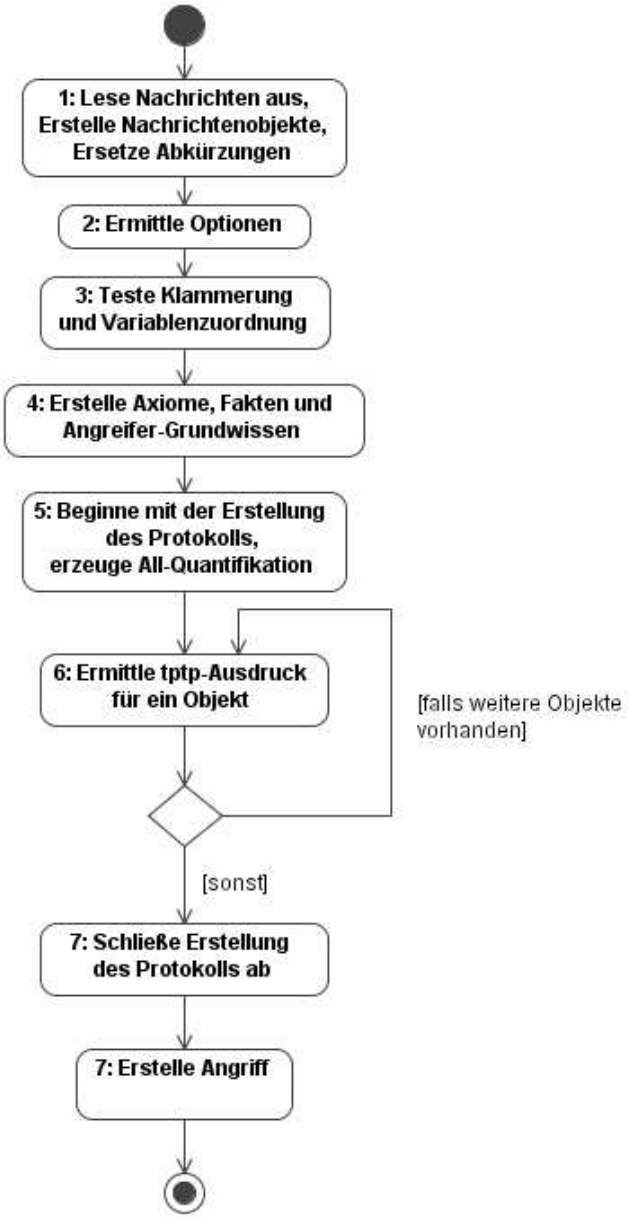

Abbildung 4.1: Algorithmus für die Erstellung der TPTP-Datei

Variablenname	**Typ**	**Inhalt**
sender	String	Name des Absender-Objekts
receiver	String	Name des Empfänger-Objekts
functionName	String	Funktionsname
message_no	Integer	Nachrichtennummer aus Diagramm
no_messages	Integer	Anzahl der Argumente dieser Nachricht
message	String	Nachricht, wie sie im Diagramm steht ohne Nachrichtennummer, eventl. einschließlich Bedingung
single_messages	Vector	Jedes Argument der Nachricht als String innerhalb eines Vectors
all_messages	String	Einzelne Argumente werden nach dem Ersetzen von Abkürzungen wieder zusammengefügt
condition	String	Komplette Bedingung, extrahiert aus Nachricht oder Tagged Value
single_condition	Vector	Bedingung, zerlegt in einzelne Bestandteile
all_conditions	String	Einzelne Bedingungen werden nach dem Ersetzen von Abkürzungen wieder zusammengefügt

Tabelle 4.1: Variablen der Klasse *SeqMsg*

und dieses entsprechend der Nachrichtennummer in ein Array einsortiert. Die Nachricht besteht aber zu diesem Zeitpunkt noch aus einem einzigen String. Um in der weiteren Bearbeitung einfacher auf die einzelnen Komponenten der Nachricht zugreifen zu können, muss der String in die einzelnen Bestandteile der Nachricht zerlegt werden. Übersicht 4.1 zeigt die einzelnen Klassenvariablen der Nachrichtenobjekte `SeqMsg`. Diese Variablen werden in diesem Schritt mit Werten gefüllt. Es wird dabei zuerst die Bedingung aus der Nachricht extrahiert (alte Guard-Notation) oder entsprechend der neuen Guard-Notation das entsprechende Tag ausgelesen und in der Variable `condition` gespeichert. Die Bedingung wird im nächsten Schritt in die Einzelteile zerlegt, falls die Bedingung aus mehreren Ausdrücken besteht, die über Konjunktionen verknüpft sind. Nachdem auch der Nachrichtenname ermittelt wurde, werden die einzelnen Argumente der Nachricht extrahiert. Auf die Einzelteile der Bedingung, sowie die einzelnen Argumente werden dann Ersetzungsregeln angewandt, indem alle Abkürzungen aus Tabelle 5.3 durch die entsprechenden TPTP-Funktionen ersetzt werden. Zum Schluß werden die Einzelteile der Bedingung, sowie die Argumente wieder zusammengesetzt in die Form, wie sie später in der TPTP-Datei erscheinen.

In einem nächsten Schritt werden dann alle Optionen ermittelt, die Einfluß auf die Erstellung der TPTP-Datei nehmen. So wird hier entschieden,

welche Variablen-Notation verwendet wird oder ob die Implikationen verschachtelt werden sollen.

Wie Aktivität 3 in Abbildung 4.1 zeigt, wird ein Test auf korrekte Klammerung der einzelnen Nachrichten durchgeführt. Wird dabei ein Fehler entdeckt, liefert das Tool z.B. folgende Fehlermeldung:

```
% Warning: Wrong number of brackets in messages:
% Message 2 contains too many closing brackets !!
```

Außerdem wird in diesem Schritt die korrekte Variablenzuordnung überprüft. Da es sich bei den Variablen, die sich auf ein bestimmtes Argument einer zuvor empfangenen Nachricht beziehen (Nachrichtenvariablen, z.B. ArgS_1_1), um lokale Variablen des Empfängers handelt, sind diese den anderen Protokollteilnehmern nicht bekannt. Kommt im Laufe des Protokolls eine Variable vor, die dem Empfänger nicht bekannt ist, erhält man einen Hinweis der Form:

```
% Note: Unknown variables in message 1: Data_C, Data_S
```

Im Schritt 4 wird die eigentliche TPTP-Datei begonnen, indem die Axiome ausgegeben werden. Diese Axiome sind im Quelltext als String festgelegt und werden einfach dem Ausgabe-String hinzugefügt. Die Fakten, welche als Tagged Values im Diagramm gespeichert sind, werden aus dem MDR-Container ausgelesen und jeweils einzeln über eine Konjunktion verknüpft. Dieser gesamte Ausdruck wird dann der Ausgabe hinzugefügt. Genauso wie die Fakten wird auch das Angreifer-Grundwissen ausgelesen, über eine Konjunktion verknüpft und ausgegeben.

Im nächsten Schritt beginnt die Modellierung des eigentlichen Protokolls. Der TPTP-Ausdruck der einzelnen Nachrichten wird dabei von einer All-Quantifikation umschlossen. In diese Quantifikation werden alle freien Variablen aufgenommen, d.h. alle Nachrichtenvariablen, die sich auf jeweils ein Argument der Nachrichten beziehen und alle Variablen, die im Protokoll vorkommen und nicht schon quantifiziert sind. Als Variablen kommen in einem Protokoll normalerweise nur Variablen vor, die sich auf Argumente beziehen (Nachrichtenvariablen) oder Variablen, die innerhalb einer Bedingung quantifiziert werden. Sollte aber doch einmal eine freie Variable vorkommen, steht diese für einen beliebigen Wert und wird somit in die All-Quantifikation aufgenommen, da innerhalb der TPTP-Datei keine freien Variablen vorkommen dürfen.

Im weiteren Ablauf wird jedes im Diagramm enthaltene Objekt, das Nachrichten verschickt oder empfängt, einzeln betrachtet und der Algorithmus in Abbildung 4.2 durchlaufen. Dabei werden für jedes Objekt alle Nachrichten des Diagramms der Reihe nach durchlaufen. Bei jeder Nachricht finden dann eine Fallunterscheidung statt. Ist das aktuelle Objekt Absender der aktuellen Nachricht, dann wird ein Ausdruck der folgenden Form erzeugt:

```
knows(Arg1) & knows(Arg2)
& (Cond1)
=> knows(conc(exp1, exp2))
```

Wobei hier `Arg1` und `Arg2` Variablen sind, die sich auf die Argumente der unmittelbar zuvor vom aktuellen Objekt empfangenen Nachricht beziehen. Es wird hiermit modelliert, dass der Empfänger beliebige Werte erhält, die nicht mit den tatsächlich abgeschickten Werten übereinstimmen müssen. Falls das Objekt vorher keine Nachricht empfangen hat, steht an der Stelle von `knows(Arg1) & knows(Arg2)` nur der Ausdruck `true`. In der nächsten Zeile wird dann die Bedingung, die vor dem Abschicken der Nachricht erfüllt sein muss, mit einer Konjunktion verknüpft. Falls keine Bedingung für diese Nachricht definiert ist, steht hier ebenfalls `true`. In der dritten Zeile werden alle Argumente der aktuellen Nachricht mit `conc()` zusammengefasst und darauf das Prädikat `knows()` angewandt. Dies dient vor allem der besseren Lesbarkeit und verändert nicht die Bedeutung des Ausdrucks, da folgende Gleichung gilt: `(knows(a) & knows(b)) = knows(conc(a,b))`.

Falls das betrachtete Objekt A Empfänger der aktuellen Nachricht ist und A eine weitere Nachricht empfängt bevor A selbst eine Nachricht verschickt, wird ein Ausdruck der folgenden Form erzeugt:

```
knows(Arg1) & knows(Arg2)
& true
=> true
```

`Arg1` und `Arg2` stehen hierbei für die Argumente der aktuellen Nachricht. Nötig ist dieser Ausdruck hier, weil der Angreifer mit dieser Nachricht auch sein Wissen erweitert, aber diese Nachricht und damit auch die Erweiterung des Angreifer-Wissens im ersten Fall nicht berücksichtigt wird.

Die einzelnen Ausdrücke, die für die ausgewählten Nachrichten im Programmablauf erstellt werden, werden entweder verschachtelt oder mit Konjunktionen verknüpft. Es wird dabei entsprechend der zu Beginn des Algorithmus ausgelesenen Option vorgegangen. Falls die Reihenfolge der Nachrichten während des Protokollablaufs eingehalten werden muss, ergibt sich ein Ausdruck dieser Form:

```
( ( knows(Arg1)
    & (Cond1) )
  => knows(exp1)
     & ( ( knows(Arg2)
           & (Cond2) )
           => knows(exp2)
       )
)
```

Wird die Reihenfolge allerdings nicht beachtet, werden die einzelnen Ausdrücke in der folgenden Form verknüpft:

```
  ( ( knows(Arg1)
      & (Cond1) )
    => knows(exp1)
  )
& ( ( knows(Arg2)
      & (Cond2) )
     => knows(exp2)
  )
```

Um wieder zum Algorithmus in Abbildung 4.1 zurückzukommen: Alle Ausdrücke, die für die einzelnen Objekte gemäß dem Algorithmus in Abbildung 4.2 erstellt wurden, werden wiederum mit Konjunktionen verknüpft. Dann wird die Erstellung des TPTP-Ausdrucks für das aktuelle Protokoll abgeschlossen, indem alle noch geöffneten Klammern geschlossen werden und der TPTP-Ausdruck mit einem Punkt beendet wird.

Im letzten Schritt des Algorithmus wird noch der Angriff erstellt. Dabei wird zuerst überprüft, ob ein Tag *secret* existiert. Falls dies der Fall ist, wird der Wert *value* des Tags *secret* innerhalb des Prädikats *knows*() verwendet. Es ergibt sich dann für dieses Beispiel folgender TPTP-Ausdruck:

```
input_formula(attack,conjecture,(
   knows(value) )).
```

Falls kein Tag *secret* existiert, wird nach eine Tag *conjecture* gesucht und dies für die Conjecture benutzt. Falls z.B. ein Tag *conjecture* mit dem Wert $equal(a, b)$ existiert, wird folgender Ausdruck in TPTP-Notation erzeugt:

```
input_formula(attack,conjecture,(
   (equal(a,b)) )).
```

Die Werte der Tags *secret* und *conjecture* werden nach Abkürzungen, wie in Tabelle 5.3 angegeben, durchsucht und diese eventuell ersetzt.

4.2 Besonderheiten bei der Implementierung

4.2.1 Aufruf von E-SETHEO

Das Tool SequenceAnalyser bietet die Möglichkeit sich das Ergebnis von E-SETHEO direkt anzeigen zu lassen. Dabei wird zuerst die TPTP-Datei erstellt und diese im temporären Verzeichnis des Betriebssystems abgespeichert. Die TPTP-Datei wird dann auf richtige Klammerung getestet und außerdem wird überprüft, ob unbekannte Variablen vorkommen. Eventuell gefundene Fehler werden über das Framework ausgegeben. Im nächsten Schritt wird E-SETHEO auf Betriebssystemebene aufgerufen. Beim Aufruf wird E-SETHEO ein Parameter für die Zeitbeschränkung mitgegeben, d.h.

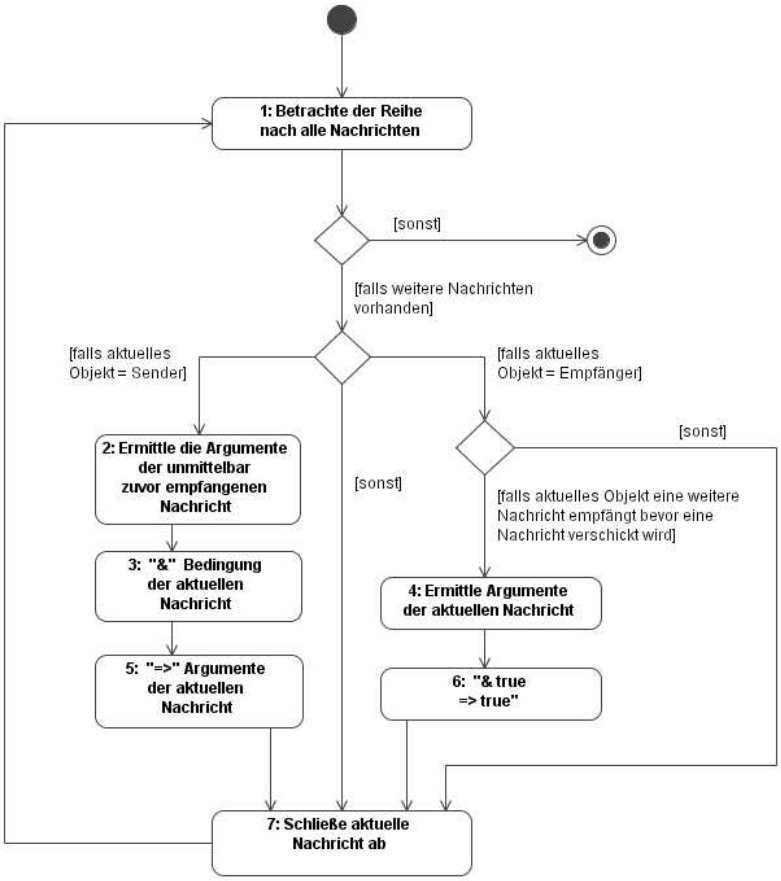

Abbildung 4.2: Algorithmus für die Erstellung des TPTP-Ausdrucks für ein bestimmtes Objekt

die Ausführung von E-SETHEO wird nach einer vorgegebenen Zeit abgebrochen. Standardmäßig sind im Framework hier 20 Sekunden definiert. Dies lässt sich im Quelltext über die Klassenvariable `maxTime` beliebig definieren. Dies bedeutet aber auch, dass es bei der Anzeige des Ergebnisses von E-SETHEO zu einer Verzögerung von eben diesen 20 Sekunden kommen kann. Die Ausgabe von E-SETHEO wird auf Betriebssystemebene abgefangen und über das Framework ausgegeben. Falls eine Ausgabe ohne Statusinformationen gewählt wurde, wird die von E-SETHEO erzeugte Ausgabe gefiltert und nur das eigentliche Ergebnis ans Framework zur Ausgabe weitergeleitet. Zum Schluß wird noch die TPTP-Datei, die im temporären Verzeichnis gespeichert wurde, gelöscht.

4.2.2 Aufruf von SPASS

Beim Aufruf von SPASS wird ähnlich vorgegangen wie beim Aufruf von E-SETHEO. Es wird zuerst die TPTP-Datei erzeugt und diese im temporären Verzeichnis gespeichert. Danach werden Tests auf richtige Klammerung und unbekannte Variablen durchgeführt. Die erzeugte TPTP-Datei muss dann in einem zusätzlichen Schritt nach DFG, der Eingabesprache von SPASS, umgeformt werden. Dies geschieht mit Hilfe des Tools *tptp2X*. Unter Linux kann dies direkt mit einem zusätzlichen Betriebssystemaufruf von tptp2X erledigt werden. Bei diesem Aufruf wird die neu erzeugte DFG-Datei ebenfalls im temporären Verzeichnis abgelegt. Unter Windows ist die Konvertierung etwas komplizierter. Dort muss zur Ausführung von tptp2X ein Prolog-Skript der folgenden Form erstellt werden:

```
:- ['c:\\programme\\TPTP-v2.7.0\\TPTP2X\\tptp2X.main'].
:- tptp2X('C:\\Temp\\protocol.tptp',100,none,dfg,'').
:- halt.
```

Dieses Prolog-Skript wird ebenfalls im temporären Verzeichnis gespeichert und dann innerhalb der SWI-Prolog-Umgebung ausgeführt. Als Ergebnis des Skripts wird die DFG-Datei im temporären Verzeichnis erstellt. Sollte während der Konvertierung ein Fehler auftreten und kann deshalb keine DFG-Datei erstellt werden, erhält der Benutzer folgende Fehlermeldung:

```
Failure: Error converting TPTP-File to DFG!
```

Falls die DFG-Datei korrekt erstellt wurde, wird im nächsten Schritt SPASS mit dieser Datei aufgerufen. Das Ergebnis von SPASS wird, wie bei E-SETHEO, abgefangen, gegebenenfalls die Statusinformationen herausgefiltert und anschließend über das Framework ausgegeben. Zum Schluß müssen hier drei Dateien gelöscht werden und zwar die TPTP-Datei, das Prolog-Skript und die DFG-Datei.

4.2.3 Text-Ausgabe

Um die Vorgaben des Frameworks einzuhalten, dürfen Ausgaben nur zeilenweise über die dafür vom Framework bereitgestellten Funktionen erfolgen. Um aber intern mit Strings arbeiten zu können, wurde die Funktion

```
displayString( String str, ITextOutput _mainOutput )
```

implementiert, welche den String `str` zeilenweise über die Standardausgabe `ITextOutput` ausgibt. Ein Zeilenende wird dabei über das Steuerzeichen `\n` erkannt. Als Parameter benötigt diese Funktion einmal den auszugebenden String `str` und eine Instanz der Standardausgabe.

4.3 Probleme während der Implementierung

4.3.1 Head-Axiome

Während der Implementierung stellte sich heraus, dass die Funktionen für den Listenzugriff, wie $fst()$, $snd()$, $thd()$ usw. in bestimmten Fällen nicht funktionieren. Wie sich nach einigen Tests herausstellte, lag das Problem am Axiom für die Funktion $head()$. Hierfür ist folgendes Axiom definiert:

```
input_formula(head_axiom,axiom,(
! [X,Y] :
( equal( head(conc(X,Y)), X ) ) )).
```

Dies bedeutet aber, dass $head()$ nur auf eine Liste mit zwei Elementen angewendet werden kann. Vor allem bei den Funktionen für den Listenzugriff wird dies dann zum Problem, da z.B. $snd()$ folgendermaßen definiert ist:

```
input_formula(snd_axiom,axiom,(
! [X] :
( equal( snd(X), head(tail(X)) ) ) )).
```

Wird nun z.B. $snd()$ auf eine Liste der Form $a :: b$ angewandt, ergibt dies $snd(a :: b) = head(tail(a :: b))$. In einem nächsten Schritt ergibt sich dann $snd(a :: b) = head(b)$. Intuitiv sollte $head(b) = b$ sein, dies kann aber obiges Axiom für $head()$ nicht berechnen, da es nur auf zwei-elementige Listen angewendet werden kann. Es kann deshalb in einer TPTP-Datei der Ausdruck $snd(a :: b)$ nicht abgeleitet werden.

Es wurden diesbezüglich dann einige verschiedene Lösungsansätze getestet, z.B. wurde mit Axiomen experimentiert, die die Funktionalität der Funktion $head()$ erweitern oder es wurde versucht Typeigenschaften für Listen und Atome einzuführen, allerdings ohne Erfolg. Es wurde dann ein Lösungsansatz entwickelt, der alle Listen der Form $a :: b$ um ein Element, das jeweils das Listenende (end of list) charakterisiert, erweitert, d.h. es wird

eine Liste $a :: b$ im Laufe der Konvertierung nach TPTP in den Ausdruck `conc(a, conc(b, eol))` übersetzt. In der Regel verändert diese Listenerweiterung die Funktionsweise des Protokolls nicht. Wird allerdings diese Ersetzung explizit nicht gewünscht, muss nur statt der Listendarstellung $a :: b$ der TPTP-Ausdruck `conc(a,b)` verwendet werden. Innerhalb des Ausdrucks `conc(a,b)` findet keine Erweiterung der Listen statt.

Kapitel 5

Bedienung

5.1 Verfügbarkeit

Das Tool SequenceAnalyser kann sowohl lokal über die GUI als auch über das Webinterface unter

`http://www4.in.tum.de:8180/vikinew/vikiweb`

benutzt werden. Erreichbar ist das Tool innerhalb des viki-Frameworks über den Menüpunkt *Sequence Analyser*. Der Funktionsumfang unterscheidet sich leicht, abhängig davon unter welchem Betriebssystem das Tool lokal benutzt wird oder ob das Webinterface Verwendung findet.

Über das Webinterface steht, außer dem Speichern der TPTP-Datei, der komplette Funktionsumfang zur Verfügung. Wird das Tool lokal unter Windows benutzt, kann nur SPASS als Theorembeweiser benutzt werden, da E-SETHEO nur für Linux erhältlich ist. Unter Linux dagegen steht der komplette Funktionsumfang zur Verfügung, sofern alle externen Tools auf dem betreffenden System installiert sind.

5.2 Funktionen

Das Tool bietet mehrere Funktionen:

Display TPTP Bei dieser Funktion wird nur die TPTP-Datei angezeigt. Diese kann dann per Hand in eine Datei kopiert werden oder direkt in die GUI eines Theorembeweisers. Diese Option kann auch bei der Fehlersuche in der TPTP-Datei hilfreich sein.

Save TPTP Die TPTP-Datei wird bei dieser Funktion im temporären Verzeichnis abgespeichert. Es wird dabei das temporäre Standard-Verzeichnis des jeweiligen Betriebssystems benutzt. (Über das Webinterface nicht möglich)

E-SETHEO	**SPASS**	**Bedeutung**	**Sicherheit des Protokolls**
proof found	proof found	Beweis für die Behauptung wurde gefunden	Protokoll unsicher
model found	completion found	Behauptung konnte nicht bewiesen werden	Protokoll sicher
resources exceeded	n/a	Zeitbeschränkung erreicht	Keine Aussage über die Sicherheit des Protokolls möglich

Tabelle 5.1: Ergebnisse der Theorembeweiser E-SETHEO und SPASS

Run E-Setheo Hier wird die fertige TPTP-Datei direkt an E-SETHEO übergeben und die Ausgabe von E-SETHEO in Kurzform über das Webinterface oder die GUI angezeigt. In der Kurzform wird nur das eigentliche Ergebnis, wie in Tabelle 5.1 aufgeführt, angezeigt. Alle Statusinformationen des Theorembeweisers werden hierbei herausgefiltert.

Run E-Setheo (with status-information) Bei diesem Kommando erfolgt die Anzeige der kompletten Ausgabe von E-SETHEO mit allen Status-Informationen.

Run SPASS Hier wird die fertige TPTP-Datei an SPASS übergeben und die Ausgabe von SPASS in Kurzform angezeigt.

Run SPASS (with status-information) Dieses Kommando liefert die komplette Ausgabe von SPASS mit Status-Informationen.

Check Variables Hiermit kann eine Überprüfung der Variablenzuordnung durchgeführt werden. Dabei wird angezeigt, welchem Objekt welche Variablen bekannt sind.

5.3 Ergebnis

Die möglichen Ergebnisse von SPASS und E-SETHEO, wie auch die Bedeutungen für die Sicherheit des Protokolls sind in Tabelle 5.1 beschrieben.

5.4 Sequenzdiagramm

Das Sequenzdiagramm muss entsprechend der Vorgaben des Frameworks mit Poseidon 1.6 erstellt werden. Aktuellere Versionen von Poseidon werden nicht unterstützt. Die mit Poseidon erstellte zargo-Datei kann direkt an das Webinterface übergeben oder lokal über die GUI eingelesen werden. Dabei ist folgende Syntax einzuhalten:

5.4.1 Beschriftung der Nachrichtenpfeile

Alle Nachrichtenpfeile müssen folgendermaßen beschriftet sein:

$$Nr : [Guard]\ msg(Arg1, Arg2, Arg3, \dots)$$

wobei

Nr:

Alle Nachrichtenpfeile müssen eine Nummer erhalten (beginnend mit 1) um sie in der korrekten Reihenfolge wieder auslesen zu können. Die Nachrichtennummer wird in Poseidon als Name behandelt, während die Bedingung und die eigentlichen Nachricht im Feld *DispatchAction* eingegeben werden kann.

Guard:

Eine Bedingung für das Abschicken der Nachricht kann in eckigen Klammern angegeben werden. Näheres zur Notation wird in Abschnitt 5.5 beschrieben.

Alternativ besteht die Möglichkeit die Bedingung in Form von *Tagged Values* anzugeben. Dies dient vor allem der besseren Lesbarkeit der Diagramme. Dabei wird ein Tag mit der Bezeichnung *guard_NR* hinzugefügt, wobei *NR* der Nachrichtennummer entspricht, zu der die entsprechende Bedingung gehört. Auf die eckigen Klammern wird in diesem Fall verzichtet.

Die Nachricht in *msg(...)* wird nur abgeschickt, wenn die Bedingung *guard* erfüllt ist. Falls keine Bedingung angegeben wurde, wird die Nachricht in jedem Fall versandt.

msg

msg steht für eine beliebige Bezeichnung der Nachricht (Nachrichtenname). Diese Bezeichnung findet nur Verwendung, falls die neue Variablen-Notation verwendet wird. Dabei wird der Nachrichtenname für die Benennung der Variablen benutzt, z.B. bezeichnet Msg_1 das erste Argument der Nachricht *msg*.

Arg1, Arg2, Arg3, ...:

Argumente (Einzelnachrichten), die zusammen an den Empfänger abgeschickt werden, falls die Bedingung *guard* erfüllt ist. Hier sind beliebig viele Argumente möglich. Näheres zur Notation siehe Abschnitt 5.5

Wird ein Argument später in einer anderen Nachricht oder Bedingung benutzt, muss folgende Notation verwendet werden:

- *Alte Notation: ArgX_a_b* steht für das b. Argument der a. Nachricht, die X empfangen hat.
  (Beispiel: *ArgS_2_4* ist das 4. Argument der 2. Nachricht die S empfangen hat)
- *Neue Notation:* Als Bezeichnung wird hier der Funktionsname der Nachricht benutzt, gefolgt von der Position des entsprechenden Arguments.
  (Beispiel: *Init_5* ist das 5. Argument der Nachricht mit dem Funktionsaufruf *Init*)

Anmerkung:

Wichtig ist bei den einzelnen Argumenten und den Bedingungen auf korrekte Klammerung zu achten.

5.4.2 Steuerung der Ausgabe mit Hilfe von Tagged Values

Während der Entwicklung des SequenceAnalyser-Tools ergaben sich mehrere Problemstellungen, die es nötig machten die Ausgabe des Tools beeinflussen zu können. Da diese Steuerung der Ausgabe am Besten mit dem Diagramm übergeben werden sollte, bieten sich dafür *Tagged Values* an. Diese Tags werden innerhalb des Diagramms immer einem Objekt zugeordnet. Beim Auslesen innerhalb des viki-Frameworks können aber nur alle Tags gemeinsam ausgelesen werden und dabei ist keine Zuordnung zu den einzelnen Objekten mehr möglich. Deshalb ist es auch prinzipiell egal, welchem Objekt die Tags zugeordnet werden. Aber der Übersichtlichkeit wegen sollten alle Tags einem Objekt *adversary* zugeordnet werden. Welche Tags hier möglich sind, zeigt die folgende Übersicht.

Angreifer-Grundwissen

Das Angreifer-Grundwissen bezeichnet das Wissen des Angreifers vor Protokollausführung. Dies kann mit Hilfe des Tags *initial knowledge* übergeben werden. Für jeden einzelnen Wert muss ein eigenes Tag benutzt werden. Es muss dabei nur der Wert selbst angegeben werden. Bei Erstellung der TPTP-Datei wird der Wert dann innerhalb des Prädikats *knows*() angegeben und alle *knows*()-Prädikate über Konjunktionen verknüpft.

Kennt der Angreifer vor Protokollausführung nur seinen eigenen öffentlichen Schlüssel k_A und seinen privaten Schlüssel k_A^{-1}, dann müssen im Diagramm zwei Tagged Values angegeben werden. Ein Tag *initial knowledge* mit dem Wert k_a und ein Tag *initial knowledge* mit dem Wert $inv(k\_a)$. Daraus wird dann folgender TPTP-Ausdruck erzeugt:

```
input_formula(previous_knowledge,axiom,(
 knows(k_a)
 & knows(inv(k_a))
 )).
```

Modellierung des Angriffs

Der Angriff steht gewissermaßen für die Hypothese, die der Theorembeweiser beweisen soll. Der Theorembeweiser versucht für diese Hypothese eine Ableitung zu finden und liefert dafür das entsprechende Ergebnis.

Soll überprüft werden, ob der Angreifer einen bestimmten Wert kennt, wird das Tag *secret* benutzt. Zur Überprüfung des Wissens eines Angreifers wird wieder das Prädikat *knows*() verwendet. Das Tag *secret* mit dem Wert *value* liefert folgenden Angriff:

```
input_formula(attack,conjecture, (
 knows(value)
 )).
```

Andererseits ist es möglich mit dem tag *conjecture* einen beliebigen Angriff in TPTP-Syntax anzugeben, z.B. der Wert *test(abc)* liefert folgenden Ausdruck:

```
input_formula(attack,conjecture, (
 test(abc)
 )).
```

Berücksichtigung der Reihenfolge der Nachrichten

Die Nachrichten innerhalb des Sequenzdiagramms werden normalerweise nur streng der Reihe nach abgearbeitet. Es kann aber auch erforderlich sein, dass die Nachrichten in einer beliebigen Reihenfolge ausgetauscht werden sollen. Über ein Tag *order* mit den Werten *true* oder *false* kann entschieden werden ob die Reihenfolge der Nachrichten berücksichtigt wird. Beim Wert *true* werden die Implikationen in der TPTP-Datei verschachtelt, d.h. die Nachrichten werden sequentiell abgearbeitet. Beim Wert *false* dagegen können die Nachrichten im Sequenzdiagramm in einer beliebigen Reihenfolge ausgetauscht werden. Standardmäßig wird die Reihenfolge der Nachrichten beachtet, d.h. wird kein Tag *order* angegeben, erhält dies automatisch den Wert *true*.

Variablen-Notation

Wie bereits in Absatz 5.4.1 auf Seite 44 beschrieben, werden zwei verschiedene Variablen-Notationen unterschieden. Mit dem Tag *notation* (und den Werten *new* oder *old*) kann zwischen der alten und neuen Notation der Variablen gewechselt werden. Standardmäßig wird die alte Notation verwendet.

Notation der Guards

Mit dem Tag *guard_notation* (und den Werten *new* oder *old*) kann zwischen der alten und neuen Notation der Guards gewechselt werden. Bei der alten Notation werden die Guards in eckigen Klammern am Beginn der Nachricht angefügt. Um eine bessere Lesbarkeit der Diagramme zu erreichen, werden die Guards in der neuen Notation als Tagged Values abgelegt. Die Tags erhalten die Bezeichnung *guard_NR*, wobei *NR* für die jeweilige Nachrichtennummer steht, zu der die Bedingung gehört. Es ist dabei egal welchem Objekt die Tags zugeordnet werden.

Fakten

Fakten sind logische Ausdrücke, die ohne Vorbedingung wahr sind. Fakten können in TPTP-Notation durch Tags mit der Bezeichnung *fact* hinzugefügt werden. Für jedes Faktum muss ein eigenes Tag benutzt werden. Alle angegebenen Fakten werden dann über Konjunktionen verknüpft.

Korrektur der Variablenschreibweise

Mit dem Tag *variables_for_dummies* und dem Wert *true* kann eine Funktion aktiviert werden, die die Schreibweise von Konstanten und Variablen korrigiert. Konstanten werden in TPTP-Notation normalerweise klein geschrieben, während Variablen groß geschrieben werden. Wird diese Funktion aktiviert, ermittelt das Tool automatisch alle Variablen und korrigiert gegebenfalls die Groß- und Kleinschreibung. Als Variablen behandelt werden alle Nachrichtenvariablen und alle in einer Bedingung vorkommenden quantifizierten Werte. Alle anderen Werte gelten als Konstanten und werden in Kleinschreibung umgewandelt. Diese Funktion dient vor allem ungeübten Anwender, die damit nicht auf die Unterscheidung von Variablen und Konstanten achten müssen. Diese Funktion ist standardmäßig nicht aktiviert.

5.4.3 Benennung von Objekten

Bei Objekten ist darauf zu achten, dass im Namen keine Leerzeichen enthalten sein dürfen. Der Objektname wird bei der alten Variablen-Notation zur Erzeugung der Variablennamen herangezogen, d.h. der Objektname darf nur Zeichen enthalten, die auch für Variablen erlaubt sind. Bei Bedarf kann auf den Objekt-Typ ausgewichen werden.

Syntax	**Bedeutung**
Wort mit kleinem Anfangsbuchstaben (z.B. *secret*)	Konstante
Wort mit großem Anfangsbuchstaben (z.B. $ArgS\_1\_1$)	Variable
f(a,b,...)	Prädikatsfunktion
~	Negation
&	Und-Verknüpfung
|	Oder-Verknüpfung
=>	Implikation
<=>	Äquivalenz
![Variable1, Variable2, ...] :	Für alle (Allquantor)
?[Variable1, Variable2, ...] :	Existenzquantor

Tabelle 5.2: TPTP-Notation

5.5 Notation innerhalb von Guards und Nachrichten

Im Sequenzdiagramm ist grundsätzlich TPTP-Notation zu verwenden. Diese ist in der Tabelle 5.2 beschrieben und eine Gegenüberstellung von UML- und TPTP-Funktionen ist in Tabelle 5.3 zu finden.

Um eine kompaktere und übersichtlichere Schreibweise im Diagramm zu erreichen, wurden für die am häufigsten verwendeten Funktionen Abkürzungen eingeführt. Diese Abkürzungen sind in Tabelle 5.3 zu finden. Alle Abkürzungen werden vom Tool bei der Erstellung der TPTP-Datei automatisch in TPTP-Notation übersetzt.

5.6 Diagramme

Dieser Abschnitt zeigt eine Übersicht verschiedener Diagramme, die das Tool SequenceAnalyser verarbeiten kann. Es wurde während der Entwicklung versucht möglichst bald eine funktionsfähige Version zur Verfügung zu stellen. In weiteren Entwicklungsstadien wurden dann inkrementell weitere Funktionen integriert. Es wurde dabei auch versucht die Diagramme lesbarer und übersichtlicher zu gestalten. Als Beispiel wurde eine Variante des TLS-Protokolls verwendet. Alle folgenden Abbildungen in diesem Abschnitt zeigen das selbe Protokoll. Es unterscheidet sich nur die Darstellung der Sequenzdiagramme.

Abbildung 5.1 zeigt die erste Version der Sequenzdiagramme. Zu Beginn wurden alle Guards innerhalb eckiger Klammern vor die eigentliche Nachricht gesetzt. Dies machte aber die Diagramme unlesbar, vor allem als Screenshot während einer Präsentation. Um die Abbildung 5.1 zu erstellen, wurde die Grafik-Export-Funktion von Poseidon benutzt.

Bedeutung	**UMLsec Operatoren**	**e-SETHEO Operatoren**	**abgekürzte Schreibweise im Diagramm**
Konkatenation	a::b	conc(a,b)	a::b
Überprüfung auf Gleichheit	a=b	equal(a,b)	a=b
Listenkopf	head(a)	head(a)	head(a)
Listenrest	tail(a)	tail(a)	tail(a)
1. Listenelement	fst(a)	fst(a)	fst(a)
2. Listenelement	snd(a)	snd(a)	snd(a)
3. Listenelement	thd(a)	thd(a)	thd(a)
4. Listenelement	frth(a)	frth(a)	frth(a)
Inverser Schlüssel	$(k)^{-1}$	inv(k)	k∧-1
asym. Verschlüsselung	$\{a\}_k$	enc(a,k)	{a}_k
asym. Entschlüsselung	$dec_k(a)$	dec(a,k)	<a>_k
sym. Verschlüsselung	$\{a\}_k$	symenc(a,k)	symenc(a,k)
sym. Entschlüsselung	$dec_k(a)$	symdec(a,k)	symdec(a,k)
Signatur	$sign_k(a)$	sign(a,k)	[a]_k
Extraktion der Signatur	$ext_k(a)$	ext(a,k)	/a_k
Hashwert	hash(a)	hash(a)	hash(a)

Tabelle 5.3: Vergleich UML- und TPTP-Operationen

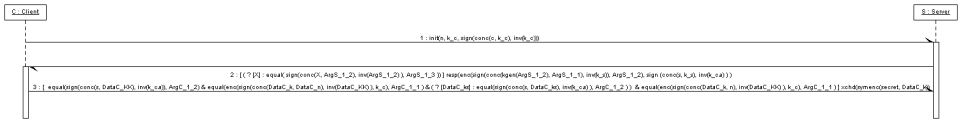

Abbildung 5.1: Variante des TLS-Protokolls als Sequenzdiagramm in der ersten Version

Es wurde dann im nächsten Schritt versucht, die Lesbarkeit der Diagramme zu erhöhen. Vor allem die Guards verbreitert die Diagrammdarstellung ziemlich stark. Als Lösung dieses Problems wurde die Möglichkeit geschaffen, die Guards als Tagged Values abzulegen. Das eigentliche Diagramm erscheint dadurch erheblich kompakter und besser lesbar. Als Beispiel dient ebenfalls eine Variante des TLS-Protokolls in Abbildung 5.2. Das Problem dabei ist allerdings, dass die Guards nicht mehr direkt innerhalb des Diagramms dargestellt werden. Es muss deshalb für die Darstellung des Diagramms auf einen Screenshot ausgewichen werden, da die Grafik-Export-Funktion von Poseidon nur das eigentliche Protokoll ohne Tags liefert.

Abbildung 5.2 zeigt die neue Variablennotation (siehe 5.4.1), die ebenfalls im Laufe der Entwicklung eingeführt wurde. Das Problem bei solchen inkrementellen Funktionserweiterungen ist aber immer, dass eventl. ältere Diagramme dann nicht mehr funktionieren. Da aber innerhalb der UMLsec-Gruppe einige Kommilitonen das Tool seit den ersten Entwicklungstufen benutzen, musste darauf geachtet werden, dass alle bereits erstellten Diagramme weiterhin problemlos funktionieren. Es wurden deshalb Tags eingeführt, die bestimmte Notationen und Funktionen steuern (siehe Abschnitt 5.4.2). So kann z.B. die Variablen-Notation mit Hilfe des Tags *notation* umgeschalten werden. Standardmäßig wird aber immer die zuerst eingeführte Notation benutzt. Abbildung 5.2 zeigt außerdem, dass zu diesem Zeitpunkt noch nicht alle Axiome eingeführt waren und deshalb innerhalb der Guards ein Pattern-Matching-Ansatz verwendet werden musste. Für die Bedingung

$$\mathtt{snd}(\mathtt{Ext}_{\mathsf{Init}_2}(\mathsf{Init}_3)) = \mathsf{Init}_2 \tag{5.1}$$

musste deshalb folgender TPTP-Ausdruck benutzt werden:

```
? [X] : equal( sign( conc( X, Init_2), inv(Init_2)), Init_3)
```

Da dies die Erstellung der Diagramme erschwert und eine Zielsetzung war, eine einfache Benutzung auch für ungeübte Anwender zu gewährleisten, wurden in einem nächsten Schritt die Axiome erweitert. Dies ermöglichte dann eine Benutzung aller wichtigen kryptographischen Funktionen. Als Problem dabei stellten sich allerdings die Axiome für den Listenzugriff heraus (siehe Abschnitt 4.3.1). Abbildung 5.3 zeigt das Sequenzdiagramm für das TLS-Protokoll in einer Form, wie es üblicherweise als UML-Diagramm dargestellt wird, mit allen kryptographischen Funktionen.

Um die Diagramme noch übersichtlicher zu gestalten und die Notation an die UMLsec-Notation anzulehnen, wurden für die gebräuchlichsten kryptographischen Funktionen Abkürzungen eingeführt (siehe Tabellen 5.3). Dabei ist es auch möglich normale TPTP-Notation mit der abgekürzten Schreibweise zu mischen, wo dies angebracht erscheint. Ein Beispiel hierzu zeigt Abbildung 5.4.

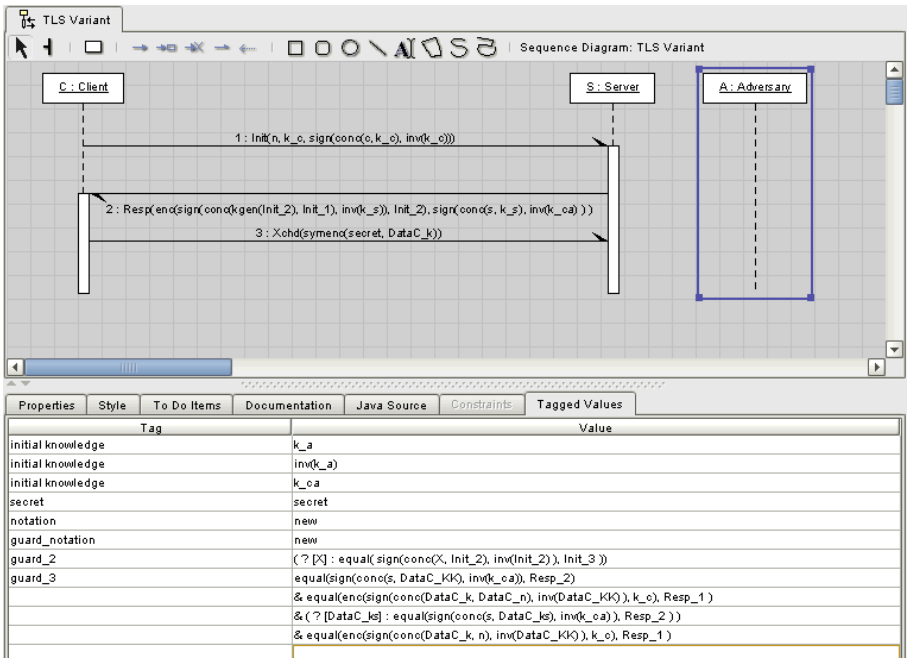

Abbildung 5.2: Variante des TLS-Protokolls mit Guards in Form von Tagged Values

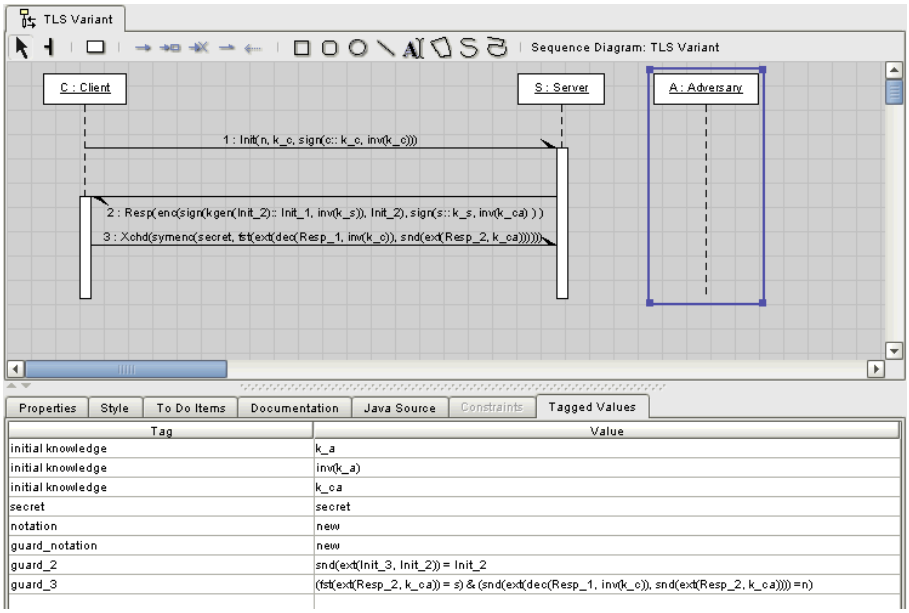

Abbildung 5.3: Variante des TLS-Protokolls mit allen Tags und kryptographischen Funktionen

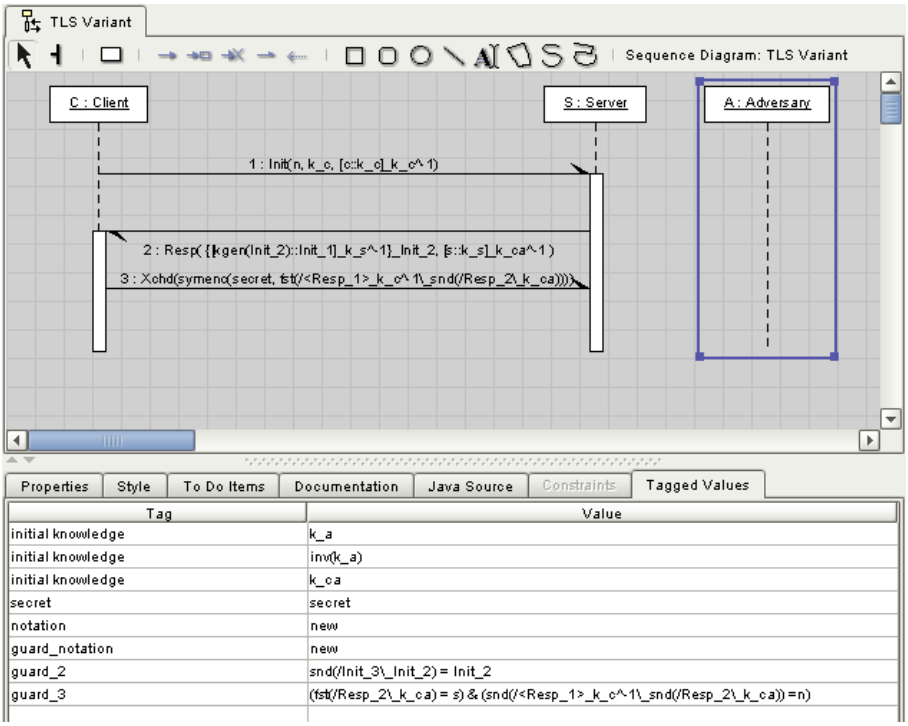

Abbildung 5.4: Variante des TLS-Protokolls mit Abkürzungen

5.7 Anbindung externer Tools

5.7.1 Anbindung von SPASS unter Windows

Für die Benutzung von SPASS unter Windows sind einige Zusatzprogramme notwendig. Dies sind im Einzelnen:

SPASS SPASS ist ebenfalls, wie E-SETHEO, ein Theorembeweiser für Logik erster Stufe mit Gleichheit. SPASS steht für verschiedene Plattformen, unter anderem für Windows und Linux, zur Verfügung. Deshalb bietet sich SPASS als Theorembeweiser zur Anbindung an das SequenceAnalyser-Tool unter Windows an, da E-SETHEO unter Windows nicht zur Verfügung steht. Die Installation erfolgt, wie unter Windows üblich, durch eine Installationsroutine.

- SPASS V2.1
- Homepage
  `http://spass.mpi-sb.mpg.de/`
- Download
  `http://spass.mpi-sb.mpg.de/download/binaries/spassdesktop21.exe`
- Standard-Installations-Verzeichnis:
  `c:\programme\SPASS\`

TPTP2X Das Tool TPTP2X dient dazu, die vom SequenceAnalyser-Tool erzeugte TPTP-Datei in das DFG-Format umzuwandeln. Das Tool steht als Linux-Version zum Download bereit. Es ist unter Windows nur in einer Prolog-Umgebung lauffähig. Es kann im Internet als zip-Archiv heruntergeladen werden. Dieses zip-Archiv muss in das unten angegebene Verzeichnis entpackt werden.

- TPTP2X 2.7.0
- Homepage
  `http://www.cs.miami.edu/~tptp/`
- Download
  `http://www.cs.miami.edu/~tptp/TPTP/Archive/`
  `TPTP-v2.7.0/TPTP-v2.7.0.tar.gz`
- Standard-Installations-Verzeichnis:
  `c:\programme\TPTP-v2.7.0\`

Zusätzlich ist noch ein temporäres Verzeichnis `c:\temp\` nötig.

Um das Tool unter Windows benutzen zu können, waren einige Anpassungen, vor allem verschiedener Pfadangaben, nötig. Geändert wurden folgende Dateien im Verzeichnis `TPTP2X`:

- tptp2X
- tptp2X.main
- tptp2X.config
- tptp2X.read

Diese geänderten Dateien stehen über die Webseite des SequenceAnalyser-Tools [Gil05] zum Download bereit. Die Original-Dateien des TPTP2X-Tools müssen mit den heruntergeladenen, geänderten Dateien überschrieben werden. Zu beachten ist allerdings, dass die geänderten Dateien nur funktionieren, wenn alle Tools in die Standard-Installationsverzeichnisse installiert wurden, da die Pfadangaben im Quelltext des TPTP2X-Tools codiert sind.

Prolog Für das Tool TPTP2X ist eine Prolog-Umgebung nötig, die ebenfalls unter Windows lauffähig ist. Ausgewählt wurde daraufhin SWI-Prolog für Windows, da dies kostenlos zur Verfügung steht. Die Installation erfolgt über eine Installationsroutine. Entwickelt und getestet wurde das Sequence-Analyser-Tool mit SWI-Prolog in der Version 5.42. Es sollten aber auch alle neueren Versionen funktionieren.

- SWI-Prolog für Windows 5.42 oder neuer
- Homepage
  `http://www.swi-prolog.org/`
- Download
  `http://gollem.science.uva.nl/cgi-bin/nph-download/SWI-Prolog/w32pl547.exe`
- Standard-Installations-Verzeichnis:
  `c:\programme\pl\`

GAWK GAWK wird für die korrekte Funktion von TPTP2X benötigt. Getestet wurde hier die Version 3.1.3-2. Es können aber auch neuere Versionen benutzt werden.

- GAWK für Windows 3.1.3-2 oder neuer
- Homepage
  `http://gnuwin32.sourceforge.net/packages/gawk.htm`
- Download
  `http://prdownloads.sourceforge.net/gnuwin32/gawk-3.1.3-2.exe`
- Standard-Installations-Verzeichnis:
  `c:\programme\GnuWin32\`

Alle Programme sollten ins Standard-Installations-Verzeichnis installiert werden. Ist dies nicht möglich, müssen die Pfadangaben im Quelltext geändert werden. Die Pfadangaben für GAWK, SWI-Prolog, TPTP2X müssen in den vier oben genannten geänderten Dateien von TPTP2X angepasst werden. Die Pfade für SPASS, TPTP2X und SWI-Prolog sind im Java-Quelltext des SequenceAnalyser-Tools codiert.

5.7.2 Anbindung unter Linux

Unter Linux besteht die Möglichkeit sowohl E-SETHEO als auch SPASS als Theorembeweiser benutzen zu können. Der Betriebssytemaufruf mit Pfadangaben ist für die externen Tools im Quellcode festgelegt, d.h. es müssen die Pfadangaben direkt im Quelltext an die Systemvorgaben angepasst werden oder es werden im Hauptverzeichnis des viki-Frameworks (Verzeichnis, das die Java-Funktion `System.getProperty("user.dir")` liefert) symbolische Links für die externen Tools angelegt. Da es schwierig ist für die verschiedenen Linux-Distributionen generelle Installationsanweisungen für die externen Tools zu geben, wird sich im Folgenden auf das Wesentliche beschränkt.

E-SETHEO

Der Theorembeweiser E-SETHEO steht auf der Seite von Dr. Gernot Stenz zum Download bereit. Diese Datei lässt sich unter Suse Linux in der Version 9.1 nach ein paar kleinen Anpassungen installieren. Genaue Anweisungen zur Installation sind in einer Readme-Datei enthalten. Es sollte nach der Installation ein symbolischer Link `run-e-setheo` angelegt werden, der auf die ausführbare Datei `run-e-setheo` von E-SETHEO verweist.

- Homepage
  `http://www4.in.tum.de/~stenzg/`
- Download
  `http://www4.in.tum.de/~stenzg/e-setheo-csp04bis.tgz`

SPASS

Für SPASS sind unter Linux im Prinzip die gleichen Tools, wie unter Windows, nötig nur mit dem Unterschied, dass die meisten Linux-Distributionen bereits eine Prolog-Umgebung sowie das Tool GAWK enthalten. Es müssen also in den meisten Fällen nur noch SPASS und TPTP2X installiert werden.

SPASS Der Theorembeweiser SPASS steht für Linux im Internet unter

`http://spass.mpi-sb.mpg.de/download/binaries/spass21pclinux.tgz`

zum Download bereit. Nachdem das Archiv entpackt wurde, kann mit dem enthaltenen Shell-Skript `configure` die Einstellungen für die Installation ermittelt werden. Die eigentliche Installation wird mit dem Aufruf

```
make install
```

durchgeführt. Die Installation ist danach abgeschlossen und die ausführbare Datei `SPASS` wurde ins Verzeichnis `/usr/local/bin` installiert. Eine detaillierte Installationsanleitung ist in der Datei `INSTALL` im heruntergeladenen Archiv zu finden. Für SPASS sollte ebenfalls, wie für E-SETHEO, ein symbolischer Link `SPASS` im Hauptverzeichnis des viki-Frameworks angelegt werden, der auf die ausführbare gleichnamige Datei des Theorembeweisers zeigt.

TPTP2X Falls das Tool TPTP2X unter Linux benutzt werden soll, muss nachdem das zip-Archiv (Download-Adresse siehe Abschnitt 5.7.1) entpackt wurde, die Datei `tptp2X_install` im Verzeichnis `TPTP2X` aufgerufen werden, um die Installation an die entsprechenden Systemvorgaben anzupassen. Es müssen dabei einige Fragen zu den Systemvoraussetzungen beantwortet werden, wie z.B. zur installierten Prolog-Umgebung und zu einigen Pfadangaben. Unter Linux kann das Tool direkt über die Shell aufgerufen werden. Hilfreich ist dabei auch ein symbolischer Link `tptp2X`, der auf die gleichnamige Datei im Verzeichnis `TPTP2X` der Installation verweist. Die Konvertierung der TPTP-Datei `protocol.tptp` nach DFG kann innerhalb der Shell mit folgendem Aufruf durchgeführt werden:

```
tptp2X -fdfg protocol.tptp
```

5.8 Fehlermeldungen

Im folgenden Abschnitt werden die Fehlermeldungen des SequenceAnalyser-Tools näher erläutert. Es wird jeweils eine beispielhafte Fehlermeldung angegeben und ihre Bedeutung erklärt. Alle Fehlermeldungen werden innerhalb der TPTP-Datei als Kommentare ausgegeben und darüber hinaus über die Standard-Ausgabe des Frameworks ausgegeben falls die TPTP-Datei nicht ohnehin angezeigt wird.

5.8.1 Falsche Klammerung

Während der Konvertierung des Sequenzdiagramms nach TPTP wird die Klammerung aller Nachrichten, wie sie im Diagramm angegeben sind, überprüft. Dabei werden alle öffnenden und schließenden Klammern gezählt und falls sich ihre Anzahl unterscheidet oder eine Klammer geschlossen wird bevor sie geöffnet wurde, wird eine Fehlermeldung der folgenden Form ausgegeben:

```
% Warning: Wrong number of brackets in messages:
% Message 2 contains too many closing brackets !!
```

Es wird dabei angegeben, welche Nachricht falsch geklammert wurde und ob die Nachricht zuviele öffnende oder schließende Klammern enthält.

Nach der Erstellung der TPTP-Datei wird diese ebenfalls auf richtige Klammerung untersucht. Es wird dann z.B. folgende Fehlermeldung ausgegeben:

```
% Error: Too many closing brackets!
```

5.8.2 Unbekannte Variablen

Alle Nachrichtenvariablen sind lokale Variablen eines Protokollteilnehmers. Alle anderen Teilnehmer kennen den Inhalt dieser Nachrichtenvariablen nicht, d.h. diese Variablen dürfen von anderen Teilnehmern nicht verwendet werden. Außerdem dürfen in der TPTP-Datei keine freien Variablen vorkommen, deshalb wird vom SequenceAnalyser-Tool eine Überprüfung auf unbekannte Variablen durchgeführt. Bei der Variablenüberprüfung werden alle Nachrichten durchlaufen und es wird überprüft, ob eine Variable dem Absender der Nachricht nicht bekannt ist weil sie vielleicht eine lokale Variable eines anderen Protokollteilnehmers ist oder es sich um eine unbekannte freie Variable handelt. Das Tool gibt dann eine Fehlermeldung der folgenden Art aus:

```
% Note: Unknown variables in guard 3: DataC_KK ; DataC_k ; DataC_n ;
% Note: Unknown variables in message 3: DataC_k ;
```

Es wird dabei der Name der unbekannten Variable angegeben und innerhalb welcher Nachricht oder Bedingung diese Variable vorkommt.

Da in der TPTP-Datei keine freien Variablen vorkommen dürfen, werden alle Variablen, die frei in Nachrichten oder Bedingungen vorkommen in die All-Quantifikation, die die Protokollmodellierung umschließt, aufgenommen. Darauf weißt auch obige Fehlermeldung hin, da freie Variablen auch als unbekannte Variablen behandelt werden.

5.8.3 Fehler beim Dateizugriff

Vom SequenceAnalyser-Tool werden zwei verschiedene Dateien gespeichert. Einmal die TPTP-Datei beim Aufruf von E-SETHEO oder SPASS und zum anderen ein Prolog-Skript für die Konvertierung von TPTP nach DFG. Kann eine dieser Datei nicht gespeichert werden, erhält man z.B. folgende Fehlermeldung:

```
Error writing tptp-File!
```

Diese Fehlermeldung weißt darauf hin, dass die TPTP-Datei nicht gespeichert werden kann. Grund hierfür kann sein, dass die Pfadangaben, die im Quellcode definiert sind, auf dem betreffenden System nicht existieren oder dass keine Berechtigung zum Schreiben von Dateien existiert.

Beim Löschen der temporären Dateien könnten ebenfalls Fehler auftreten, dann wird eine Fehlermeldung der folgenden Form ausgegeben:

```
Error deleting File C:\Temp\test_237244647.tptp !
```

Tritt bei einem Betriebssystemaufruf, wie dem Aufruf von E-SETHEO, ein Fehler auf, wird direkt der Fehler des Betriebssystems über die Standardausgabe ausgegeben.

Alle Fehler dieses Abschnitts werden nicht innerhalb der TPTP-Datei ausgegeben, sondern direkt über die Standardausgabe des Frameworks.

5.8.4 Fehler bei der Konvertierung von TPTP nach DFG

Tritt bei der Konvertierung der TPTP-Datei nach DFG mit Hilfe des Tools TPTP2X ein Fehler auf, so wird keine DFG-Datei erzeugt und das Sequence-Analyser-Tool gibt folgende Fehlermeldung aus:

```
Failure: Error converting TPTP-File to DFG!
```

Es läßt sich leider keine detailliertere Fehlermeldung angeben, da die Konvertierung unter Windows in einer Prolog-Umgebung abläuft und da diese Prolog-Umgebung als eigenständige Windows-Anwendung mit eigener GUI gestartet wird, können die Fehlermeldungen nicht abgefangen werden um sie über das Framework auszugeben. Es kann aber vielleicht helfen, die TPTP-Datei abzuspeichern und dann außerhalb des Tools mit TPTP2X zu konvertieren. Der Aufruf für die Konvertierung von TPTP nach DFG mit TPTP2X ist in Abschnitt 5.7.2 zu finden.

5.9 Webinterface

Das Webinterface des viki-Frameworks ist im Internet über die Adresse

```
http://www4.in.tum.de:8180/vikinew/vikiweb
```

zu erreichen. Wird diese Adresse in einem Browser aufgerufen, erscheint die Startseite des Webinterface (siehe Abbildung 5.5). Dort kann im Feld *Model file* die zargo-Datei von Poseidon angegeben werden. Alternativ kann über den Button *Durchsuchen* die Datei ausgewählt werden. Außerdem muß das Tool *Sequence Analyser* über einen Button ausgewählt werden. Danach wird das Ganze mit dem Button *Submit* bestätigt.

Abbildung 5.6 zeigt die Auswahl der Funktionen. Diese Seite erscheint, wenn die Startseite des Webinterface mit dem Button *Submit* abgeschickt wurde. Auf dieser Seite kann nun die Funktion des SequenceAnalyser-Tools ausgewählt werden. Eine Übersicht der Funktionen ist in Abschnitt 5.2 zu finden. Wird eine Funktion ausgewählt, muss die Auswahl wieder mit dem Button *Submit* bestätigt werden.

Wurde mit *Submit* bestätigt, erscheint auf der nächsten Seite unterhalb der Funktionsübersicht das Ergebnis (siehe Abbildung 5.7). Soll auf dem selben Diagramm eine weitere Funktion ausgeführt werden, kann diese Funktion direkt ausgewählt und über den Button *Submit* abgeschickt werden. Möchte man allerdings ein neues Diagramm überprüfen, erreicht man über den Link *Reset* wieder die Startseite des Webinterface um ein neues Diagramm zu laden.

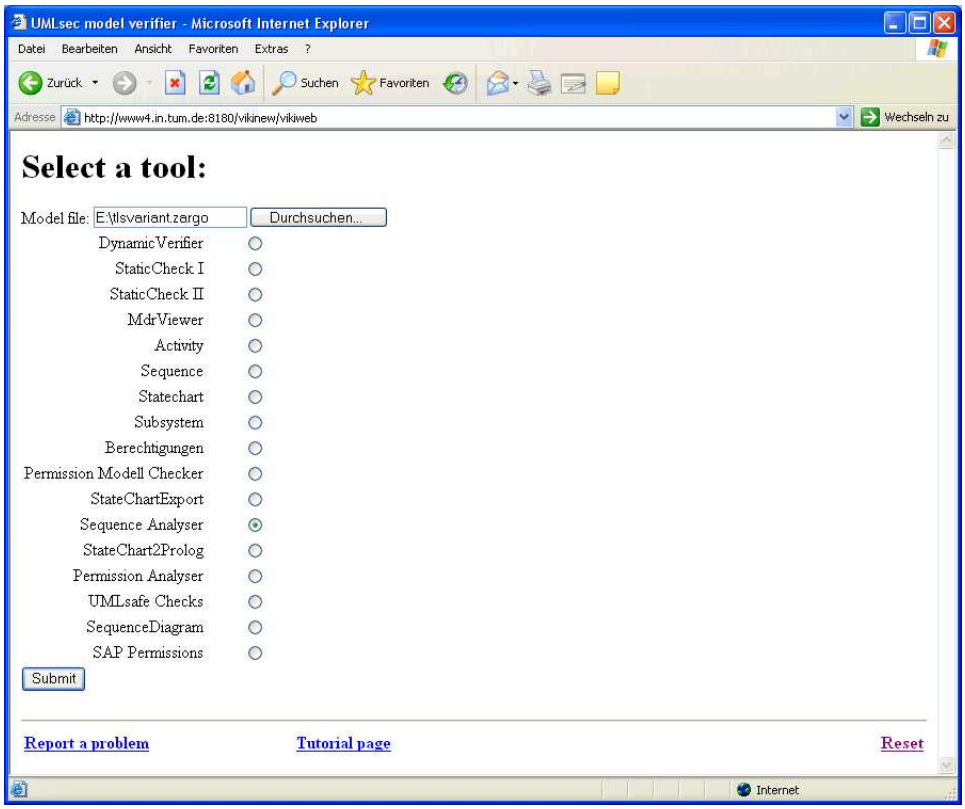

Abbildung 5.5: Startseite des Webinterface

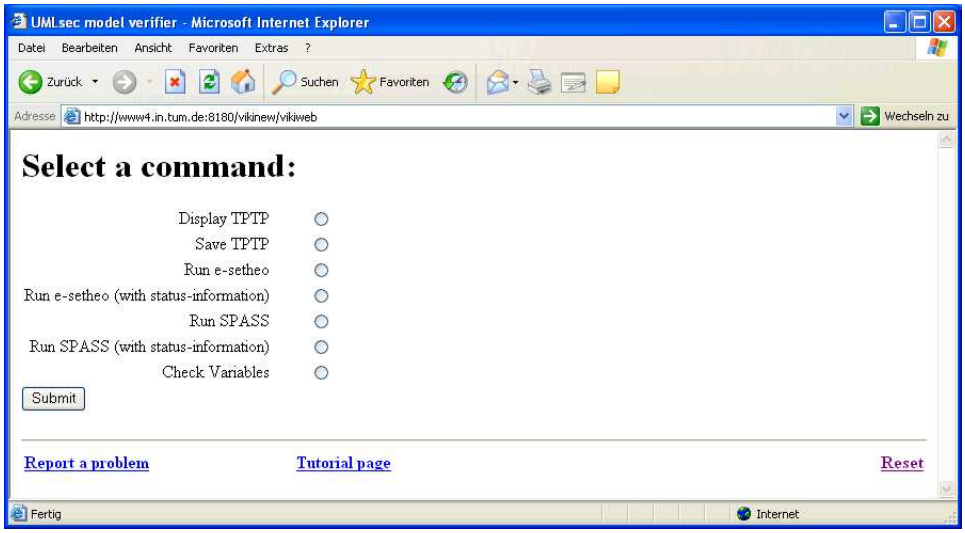

Abbildung 5.6: Auswahl der Funktionen im Webinterface

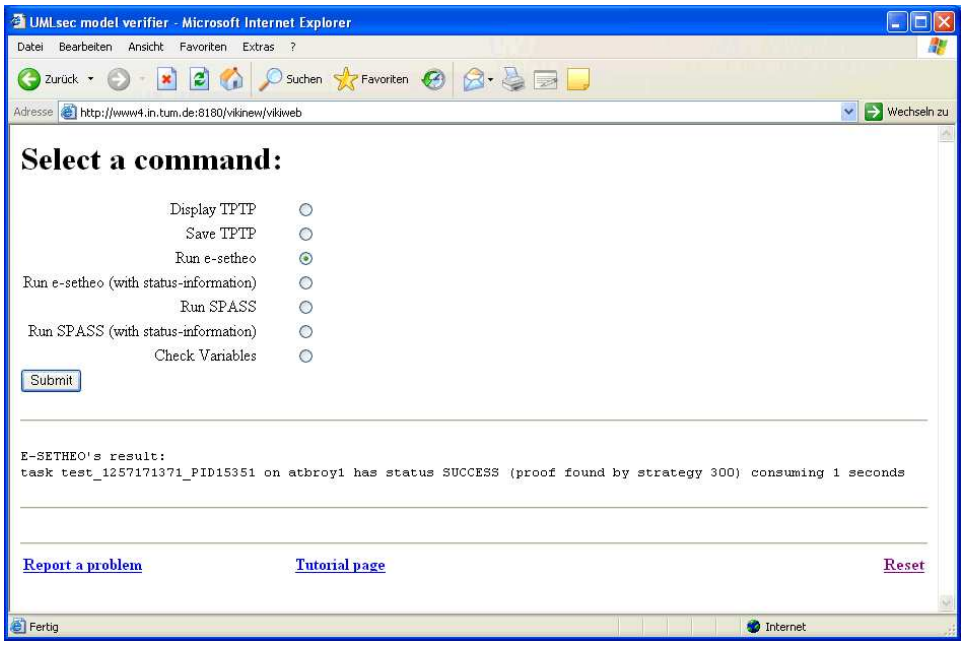

Abbildung 5.7: Webinterface mit Ergebnis von E-SETHEO

5.10 Benutzeroberfläche

Abbildung 5.8 zeigt die Benutzeroberfläche des viki-Frameworks. Innerhalb der GUI kann über den Menüpunkt $File \rightarrow Load$ ein Diagramm in Form einer zargo-Datei geladen werden. Anschließend sollte über den Menüpunkt $Window \rightarrow SequenceAnalyser$ in das Ausgabefenster des SequenceAnalyser-Tools gewechselt werden. Dann kann über das Menü

$$Tool \rightarrow SequenceAnalyser$$

eine Funktion ausgewählt werden. Die Funktionen des Tools sind in Abschnitt 5.2 erläutert. Die Ausgaben des SequenceAnalyser-Tools werden im oberen Fenster der GUI ausgegeben, während Statusmeldungen des Frameworks im unteren Fenster erscheinen.

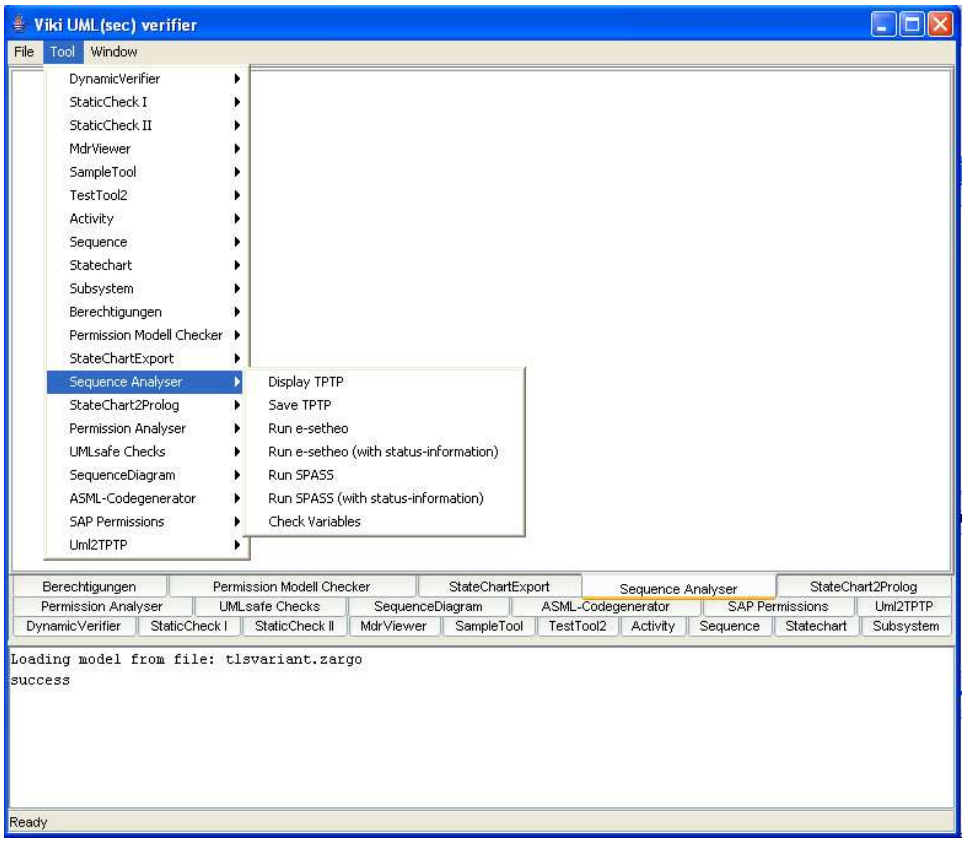

Abbildung 5.8: GUI des viki-Frameworks

Kapitel 6

Fallbeispiel: Variante des TLS-Protokolls

Das TLS[1]-Protokoll ist der Nachfolger des Internet-Protokolls SSL. Im Folgenden wird eine Variante des TLS-Protokolls auf seine Sicherheit untersucht. Ein UML-Sequenzdiagramm dieses Protokolls zeigt Abbildung 6.1. Ziel des Protokolls ist es einen sicheren Kanal über eine unsichere Datenverbindung herzustellen um Datengeheimhaltung und Server-Authentizität zu gewährleisten. Aufgabe soll es in diesem Abschnitt sein, das Protokoll daraufhin zu untersuchen, ob es einem Angreifer, wie in Abschnitt 2.1 beschrieben, möglich ist die Datengeheimhaltung zu umgehen.

Protokollablauf

Teilnehmer des Protokolls sind einmal der Client C sowie ein Server S. Der Client kennt vor Protokollausführung den Namen des Servers, während der Server nicht in Besitz des Client-Namens ist, da es nicht Ziel des Protokolls ist Client-Authentizität sicherzustellen. Im Folgenden wird der Einfachheit halber nur ein Durchlauf des Protokolls behandelt. Normalerweise behandelt ein Server mehrere Protokollabläufe mit unterschiedlichen Clients, während auch der Client das Protokoll des öfteren mit unterschiedlichen Servern durchläuft. Der Client-Name C sowie der Server-Name S sind beliebige Instanzen der Client- und Server-Objekte.

Im Folgenden wird von einem Standard-Angreifer ausgegangen, welcher die Datenverbindung zwischen dem Client und Server kontrolliert und somit alle Nachrichten, die zwischen den Protokollteilnehmern ausgetauscht werden, lesen kann. Dies bedeutet aber, dass alle Nachrichten, die im Netzwerk verschickt werden, in das Wissen des Angreifers übergehen. Außerdem ist es dem Angreifer möglich, Nachrichten zu löschen und neue Nachrichten zu verschicken. Daraus folgt, dass der Angreifer somit das Protokoll mit dem

[1] Transport Layer Security

Client oder dem Server ausführen kann und sich jeweils als der andere Protokollteilnehmer (Server oder Client) ausgeben kann.

Der Client C, sowie der Server S besitzen einen öffentlichen Schlüssel K_C sowie K_S mit den dazugehörigen privaten Schlüsseln K_C^{-1} und K_S^{-1}. Der Server S benutzt ein Zertifikat $Sign_{K_{CA}^{-1}}(S :: K_S)$, welches von einer Zertifizierungsinstanz signiert wurde und den Servernamen sowie dessen öffentlichen Schlüssel enthält. Der Client besitzt den öffentlichen Schlüssel der Zertifizierungsinstanz K_{CA}, um durch Überprüfung des Server-Zertifikats die Integrität der Nachrichten sicherstellen zu können. Die Nonce N wird vom Client frisch generiert. Der sichere Kanal, der durch das Protokoll aufgebaut werden soll, dient zur sicheren Übertragung des Wertes s vom Client zum Server. Während des Protokollablaufs wird der Sitzungs-Schlüssel k vom Server frisch generiert. Alle frisch generierten Daten, wie Sitzungsschlüssel und Nonce sind unabhängig von zuvor durchgeführten Protokollabläufen.

Einen vereinfachten Protokollablauf mit allen sicherheitsrelevanten Nachrichten und Daten zeigt Abbildung 6.1. Im abgebildeten Diagramm kennzeichnet die Bezeichnung msg_n das $n$te Argument der Nachricht msg. Außerdem werden Ersetzungsregeln der Form $ab ::= msg_n$ verwendet. Da es sich im Folgenden um einen speziellen Protokollablauf handelt, erhalten die frisch generierten Werte (Nonce N und Sitzungsschlüssel k) sowie das Geheimnis s einen Index, um zu verdeutlichen, dass diese nur in diesem Protokollablauf gültig sind. Ein spezieller Protokollablauf sieht folgendermaßen aus: Der Client beginnt das Protokoll indem er die Nachricht

$$\mathsf{Init}(\mathsf{N_i}, \mathsf{K_C}, \mathtt{Sign}_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$$

an den Server schickt. Danach überprüft der Server die Bedingung

$$\left[\mathtt{snd}(\mathtt{Ext}_{\mathsf{K'_C}}(\mathsf{c_C})) = \mathsf{K'_C}\right]$$

wobei $\mathsf{c_C} ::= \mathsf{init_3}$ und $\mathsf{K'_C} ::= \mathsf{init_2}$, d.h. es wird überprüft, ob der Schlüssel K_C in der Signatur mit dem im Klartext übertragenen Schlüssel übereinstimmt. Ist die Bedingung erfüllt, schickt der Server die Nachricht

$$\mathsf{Resp}(\{\mathtt{Sign}_{\mathsf{K_S^{-1}}}(\mathsf{k_i} :: \mathsf{N'})\}_{\mathsf{K'_C}}, \mathtt{Sign}_{\mathsf{K_{CA}^{-1}}}(\mathsf{S} :: \mathsf{K_S}))$$

zurück zum Client, wobei $\mathsf{N'} ::= \mathsf{init_1}$ und $\mathsf{K'_C} ::= \mathsf{init_2}$. Diese Nachricht enthält den aktuellen Sitzungsschlüssel k_i sowie das Zertifikat $\mathtt{Sign}_{K_{CA}^{-1}}(S :: K_S)$ des Servers um Server-Authentizität sicherzustellen. Daraufhin untersucht der Client die Bedingung

$$\left[\mathtt{fst}(\mathtt{Ext}_{\mathsf{K_{CA}}}(\mathsf{c_S})) = \mathsf{S} \wedge \mathtt{snd}(\mathtt{Ext}_{\mathsf{K'_S}}(\mathtt{Dec}_{\mathsf{K_C^{-1}}}(\mathsf{c_k}))) = \mathsf{N_i}\right]$$

wobei $\mathsf{c_k} ::= \mathsf{resp_1}$, $\mathsf{c_S} ::= \mathsf{resp_2}$ und $\mathsf{K'_S} ::= \mathtt{snd}(\mathtt{Ext}_{\mathsf{K_{CA}}}(\mathsf{c_S}))$. Dies bedeutet, der Client untersucht die Echtheit des Zertifikats und außerdem überprüft

er, ob die korrekte Nonce der ersten Nachricht zurückgeschickt wurde. Ist diese Bedingung erfüllt, beginnt C mit der verschlüsselten Übertragung des Geheimnisses s_i, indem er die Nachricht

$$\mathsf{Xchd}(\{\mathsf{s_i}\}_{\mathsf{k'}})$$

abschickt, wobei $\mathsf{k'} ::= \mathtt{fst}(\mathtt{Ext}_{\mathsf{K'_S}}(\mathtt{Dec}_{\mathsf{K_C^{-1}}}(\mathsf{c_k})))$. Falls eine der Bedingungen nicht erfüllt sein sollte, wird der Protokolldurchlauf abgebrochen.

Eine vereinfachte Darstellung liefert folgendes Schema:

$$\begin{aligned} C \rightarrow S &: \mathsf{N_i}, \mathsf{K_C}, \mathtt{Sign}_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}) \\ S \rightarrow C &: \{\mathtt{Sign}_{\mathsf{K_S^{-1}}}(\mathsf{k_i} :: \mathsf{N'})\}_{\mathsf{K'_C}}, \mathtt{Sign}_{\mathsf{K_{CA}^{-1}}}(\mathsf{S} :: \mathsf{K_S}) \\ C \rightarrow S &: \{\mathsf{s_i}\}_{\mathsf{k'}} \end{aligned}$$

Bei dem vorliegenden Angreifer-Szenario ist allerdings nicht sichergestellt, dass eine Nachricht, welche von C an S geschickt wird, auch unverändert bei S ankommt. Das Gleiche gilt für Nachrichten vom Server zum Client. Für den Angreifer besteht die Möglichkeit einzelne Werte in die Nachrichten einzuschleusen und ursprüngliche Werte zu ersetzen, falls er die dafür nötigen Ver- und Entschlüsselungsverfahren beherrscht und die entsprechenden Schlüssel besitzt.

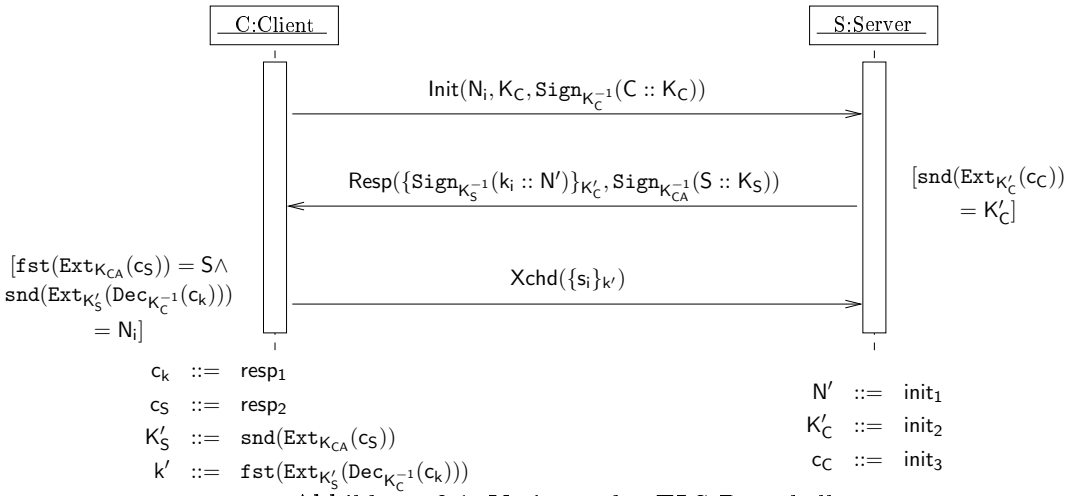

Abbildung 6.1: Variante des TLS-Protokolls

Überprüfung der Sicherheit

Das vorliegende Protokoll wird nun auf Sicherheitsaspekte mit Hilfe des Tools SequenceAnalyser überprüft. Ziel des Protokolls ist die Geheimhaltung von s_i sicherzustellen, d.h. s_i darf nicht in das Wissen des Angreifers übergehen. Um diesen Sachverhalt mit Hilfe des in Kapitel 3 erläuterten Prädikats *knows()*

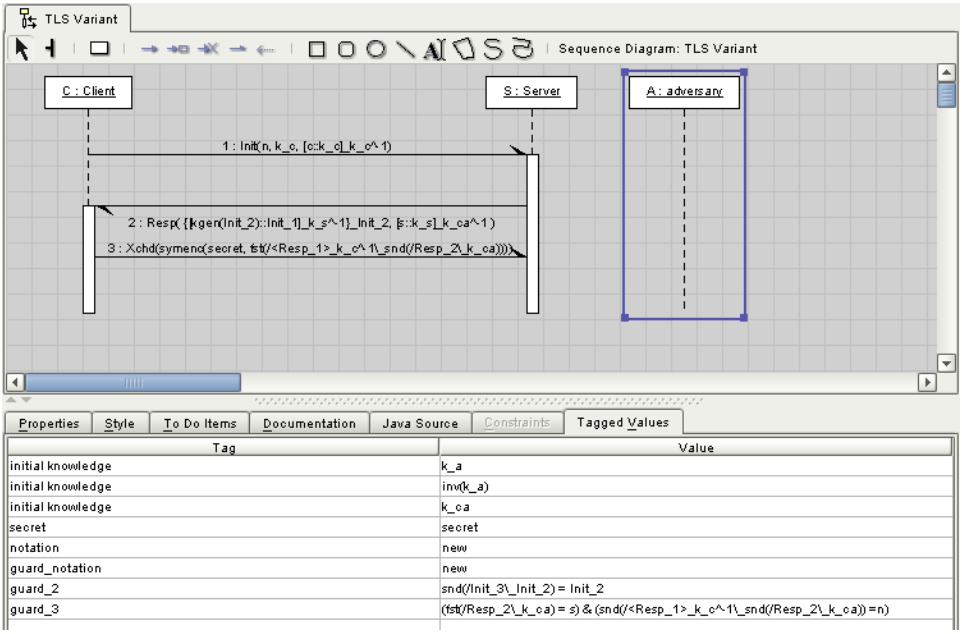

Abbildung 6.2: Variante des TLS-Protokolls gezeichnet mit Poseidon als Eingabe für den SequenceAnalyser

auszudrücken, darf $knows(s_i)$ nicht erfüllt sein. Die komplette TPTP-Datei, die vom SequenceAnalyser aus dem Diagramm in Abbildung 6.2 erzeugt wurde, ist in Anhang B.1.1 zu finden. Wird mit dieser TPTP-Datei E-SETHEO aufgerufen, erhält man folgendes Ergebnis:

```
E-SETHEO's result:

task test_943905762_PID26561 on atbroy1 has status SUCCESS
  (proof found by strategy 300) consuming 1 seconds
```

SPASS liefert folgendes Ergebnis:

```
-------------------------SPASS-START-----------------------------
SPASS V 2.1
SPASS beiseite: Proof found.
SPASS derived 484 clauses, backtracked 0 clauses and kept 238 clauses.
SPASS allocated 660 KBytes.
SPASS spent0:00:00.32 on the problem.
-------------------------SPASS-STOP------------------------------
```

Dies bedeutet, laut Tabelle 5.1, dass eine Ableitung für die Conjecture gefunden wurde, also dass der Angreifer das Geheimnis s_i kennt und die Geheimhaltung im vorliegenden Protokoll, beim angenommenen Standard-Angreifer, nicht gewährleistet ist. Durch die Überprüfung mit E-SETHEO

und SPASS steht zwar fest, dass ein Angriff existiert, aber der Angriff selbst muss mit anderen Werkzeugen ermittelt werden.

Folgender Angriff ist [Jür04] entnommen:

$$
\begin{array}{rcl}
 & \xrightarrow{\mathsf{N_i}::\mathsf{K_C}::\mathtt{Sign}_{\mathsf{K_C}^{-1}}(\mathsf{C}::\mathsf{K_C})} & \\
C & & A \xrightarrow{\mathsf{N_i}::\mathsf{K_A}::\mathtt{Sign}_{\mathsf{K_A}^{-1}}(\mathsf{C}::\mathsf{K_A})} S \\
 & \xleftarrow{\{\mathtt{Sign}_{\mathsf{K_S}^{-1}}(\mathsf{k_i}::\mathsf{N'})\}_{\mathsf{K'_C}},\mathtt{Sign}_{\mathsf{K_{CA}}^{-1}}(\mathsf{S}::\mathsf{K_S})} & \\
S & & A \xleftarrow{\{\mathtt{Sign}_{\mathsf{K_S}^{-1}}(\mathsf{k_i}::\mathsf{N'})\}_{\mathsf{K'_A}},\mathtt{Sign}_{\mathsf{K_{CA}}^{-1}}(\mathsf{S}::\mathsf{K_S})} S \\
 & \xrightarrow{\{\mathsf{s_i}\}_{\mathsf{k'}}} & \\
C & & A \xrightarrow{\{\mathsf{s_i}\}_{\mathsf{k'}}} S
\end{array}
$$

Verbesserte Version des TLS-Protokolls

In [Jür04] wird vorgeschlagen den Ausdruck $k_i :: N'$ in der Nachricht *Resp* durch den Ausdruck $k_i :: N' :: K_C$ zu ersetzen, sowie eine Bedingung hinzuzufügen, die den neu hinzugefügten Teil der Nachricht überprüft. Der öffentliche Schlüssel K_C wird hier stellvertretend für die Identität des Clients benutzt, ebenso könnte hier der Clientname C verwendet werden. Es entsteht dann folgender Nachrichtenfluß:

$$
\begin{array}{lcl}
C \rightarrow S & : & \mathsf{N_i}, \mathsf{K_C}, \mathtt{Sign}_{\mathsf{K_C}^{-1}}(\mathsf{C} :: \mathsf{K_C}) \\
S \rightarrow C & : & \{\mathtt{Sign}_{\mathsf{K_S}^{-1}}(\mathsf{k_i} :: \mathsf{N'} :: \mathsf{K_C})\}_{\mathsf{K'_C}}, \mathtt{Sign}_{\mathsf{K_{CA}}^{-1}}(\mathsf{S} :: \mathsf{K_S}) \\
C \rightarrow S & : & \{\mathsf{s_i}\}_{\mathsf{k'}}
\end{array}
$$

Das Zertifikat in der ersten Nachricht ist nur ein selbst-signiertes Zertifikat, dadurch wird keine Client-Authentizität sichergestellt, d.h. ein Angreifer kann dieses Zertifikat selbst herstellen und seinen öffentlichen Schlüssel als Schlüssel des Clients ausgeben, wie in obigem Angriff beschrieben. In der darauf folgenden Nachricht des Servers ist der öffentliche Schlüssel des Angreifers in einem vom Server signierten Zertifikat enthalten. Dem Angreifer ist es nicht möglich dieses Zertifikat zu verändern. Er kann dieses Zertifikat nur unverändert an den Client weiterleiten. Der Client überprüft aber vor dem Abschicken der dritten Nachricht den enthaltenen Schlüssel und stellt daraufhin fest, dass es sich nicht um seinen eigenen Schlüssel handelt und bricht deshalb die Protokollausführung ab.

Das veränderte Protokoll als UML-Sequenzdiagramm zeigt Abbildung 6.3. Das in Poseidon gezeichnete Diagramm in Abbildung 6.4 wurde für das Tool SequenceAnalyser angepasst. E-setheo liefert bei Überprüfung des Protokolls auf die Geheimhaltung des Geheimnisses s_i folgende Ausgabe:

```
E-SETHEO's result:
task test_16088717_PID26640 on atbroy1 has status SUCCESS
  (model found by strategy 300) consuming 1 seconds
```

SPASS liefert bei dem Diagramm in Abbildung 6.4 folgendes Ergebnis:

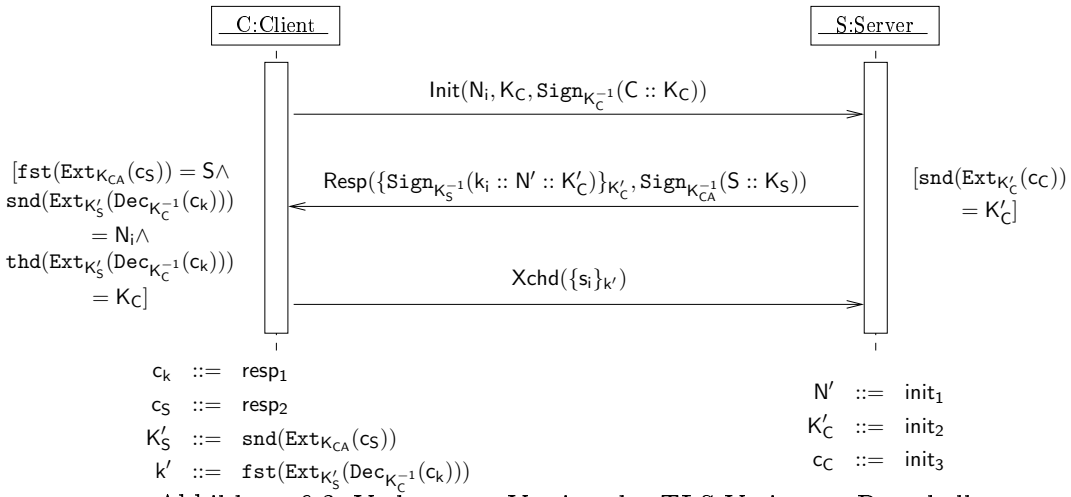

Abbildung 6.3: Verbesserte Version des TLS-Varianten-Protokolls

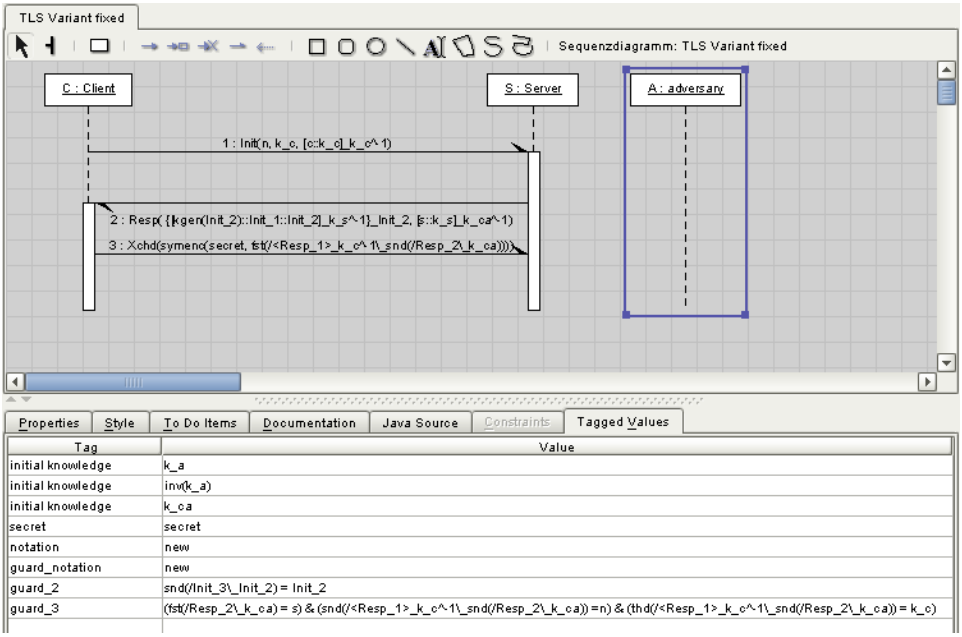

Abbildung 6.4: Korrigierte Version des TLS-Varianten-Protokolls, gezeichnet mit Poseidon als Eingabe für den SequenceAnalyser

```
-------------------------SPASS-START-----------------------------
SPASS V 2.1
SPASS beiseite: Completion found.
SPASS derived 742 clauses, backtracked 0 clauses and kept 292 clauses.
SPASS allocated 695 KBytes.
SPASS spent 0:00:00.31 on the problem.
-------------------------SPASS-STOP------------------------------
```

Diese Ausgabe von E-SETHEO und SPASS bedeutet nun, dass die Geheimhaltung des Geheimnisses s_i gewährleistet ist. Diese Aussage kann allerdings dem korrigierten Protokoll keine generelle Sicherheit bestätigen. Dieses Ergebnis bezieht sich nur auf das zugrundeliegende Angreifermodell. Vor allem wird davon ausgegangen, dass keine Werte wie z.B. das Geheimnis s_i, die privaten Schlüssel des Clients und des Servers K_C^{-1}, K_S^{-1}, Zertifikate des Servers und Clients oder der Sitzungs-Schlüssel k_i sich vor Protokollausführung im Wissen des Angreifers befinden. Außerdem ist wichtig, dass der Angreifer nicht aus einer vorhergehenden Protokollausführung den Nachrichtenteil $\{Sign_{K_S^{-1}}(k_i :: N_i :: K_C)\}_{K_C}$ kennt, sonst würde die Protokollverbesserung nicht greifen und der oben beschriebene Angriff wäre weiterhin möglich.

Kapitel 7

Zusammenfassung

Das Ergebnis der vorliegenden Arbeit ist ein Tool, das Sequenzdiagramme auf Sicherheitslücken überprüfen kann. Das Tool konnte seine Einsatzfähigkeit bereits innerhalb der UMLsec-Gruppe unter Beweis stellen. Einige Kommilitonen verwendeten das Tool bereits für ihre eigene Diplomarbeit oder innerhalb eines Systementwicklungsprojekts. Es zeigte sich dabei, dass es möglich ist, das Tool ohne weitreichende Kenntnis des Hintergrunds benutzen zu können. Da diese Ausarbeitung erst zum Ende der Arbeit fertig gestellt wurde, stand den meisten Benutzern nur eine kurze Anleitung und Einführung mit Beschreibung der Diagrammsyntax im Internet zur Verfügung. Es zeigt sich also, dass nur eine kurze Einführung nötig ist um das Tool in der Praxis benutzen zu können. Es wurde dabei bereits innerhalb der Arbeit [Jür05] eine Sicherheitslücke entdeckt, die bisher noch nicht bekannt war. Es zeigt sich dabei auch, dass das SequenceAnalyser-Tool effizient genug ist, um Protokolle in einer normalen Größenordung, wie sie in der Industrie vorkommen, bearbeiten zu können. Das Ziel dem Benutzer ein Tool zur Verfügung zu stellen, das ein Diagramm automatisch analysiert und dem Benutzer eine Antwort liefert, ob seine vorgegebene Sicherheitsanforderung erfüllt wird oder nicht, konnte mit dem entwickelten Tool ebenfalls erfüllt werden. Das Tool steht über das Webinterface des viki-Frameworks zur Verfügung und bietet dem Nutzer so die Möglichkeit ohne Installation und ohne Rücksichtnahme auf irgendwelche Systemanforderungen seine Diagramme einfach überprüfen zu können.

Darüber hinaus wurde innerhalb der UMLsec-Gruppe in einer anderen Diplomarbeit ([Kop05]) ein ähnlicher Ansatz zur Sicherheitsüberprüfung von Diagrammen in Prolog implementiert. Dieser Ansatz in Prolog ist zwar nicht so effizient, wie die Überprüfung mit automatischen Theorembeweisern, bietet aber die Möglichkeit über die Variablenbelegung direkt einen Angriff anzugeben. Dieses Tool bietet sich also an, falls für ein Diagramm mit dem SequenceAnalyser eine Angriffsmöglichkeit festgestellt wurde, einen konkreten Angriff mit Prolog zu ermitteln. Diese beiden Tools erweitern sich somit

in idealer Weise.

Natürlich finden sich im entwickelten SequenceAnalyser-Tool noch Erweiterungsmöglichkeiten. So wird bereits in einer anderen Diplomarbeit ein Ansatz entwickelt, die Berechnung des Theorembeweisers effizienter zu gestalten. In dieser Arbeit sollen z.B. Vorkommen von $ext()$ (Extraktion einer Signatur) durch einen Pattern-Matching Ansatz mit der Signatur-Funktion $sign()$ ersetzt werden (Beispiel siehe Abschnitt 5.6). Dieses Vorgehen bei der Erstellung der TPTP-Datei beschleunigt die Berechnung des Theorembeweisers.

Literaturverzeichnis

[FS00] Martin Fowler, Kendall Scott. UML konzentriert. Addison Wesley, 2000.

[Gil05] Andreas Gilg. Anleitung und Tutorial zum SequenceAnalyser Tool. `http://www4.in.tum.de/~umlsec/Internals/AndreasG`

[GL04] Susanne Graf, Wolfgang Linsmeier. TPTP Tutorial v1.0. UMLsec-Gruppe, 2004

[JK04] Jan Jürjens, Thomas A. Kuhn. Automated Theorem Proving for Cryptographic Protocols with Automatic Attack Generation. 2004.

[JS04] Jan Jürjens, Pasha Shabalin. Automated Verification of UMLsec Models for Security Requirements. 2004.

[Jür04] Jan Jürjens. Secure Systems Development with UML. Springer, März 2004.

[Jür05] Jan Jürjens. Automated Security Analysis of Biometric Authentication Protocols using First-Order Logic. 2005.

[Kop05] Dimitri Kopjev. Werkzeugunterstützte Sicherheitsanalyse von Workflows bezüglich SAP Berechtigungen. Diplomarbeit, Technische Universität München, 15.02.2005.

[Sch04] Robert Schmidt. Modellbasierte Sicherheitsanalyse mit UMLsec: ein Biometrisches Zugangskontrollsystem. Diplomarbeit, Ludwig-Maximilians Universität München, 26.4.2004.

[Sch03] Aidong Schwaiger. Grundlagen für werkzeugunterstützte Analyse von Geschäftsprozessen und SAP Berechtigungen. Systementwicklungsprojekt, Technische Universität München, 2004.

[Sch04a] Stephan Schwarzmüller. Werkzeuggestützte Analyse von UML-Subsystemen und Sequenzdiagrammen. Diplomarbeit, Technische Universität München, 16.02.2004.

[Sha04] Pavlo Shabalin. Model Checking UMLsec. Diplomarbeit, Technische Universität München, 15.08.2004.

[SS04] Christian B. Suttner, Geoff Sutcliffe. The TPTP Problem Library - TPTP v2.6.0. Department of Computer Science, University of Miami, 2004. http://www.cs.miami.edu/~tptp/

[Spa04] SPASS Home Page. http://spass.mpi-sb.mpg.de/

[Ste02] Gernot Stenz. Theorembeweisen. Vortrag, Technische Universität München, SS 2002.

[Yua04] Jun Yuan. Analyse eines Purchase Protokolls mit E-SETHEO. Systementwicklungsprojekt, Technische Universität München, 2004.

[Uml05] UMLsec-Gruppe. UMLsec Internals. http://www4.in.tum.de/~umlsec/Internals

[Vik05] Viki-Framework Downloadpage http://www4.in.tum.de/~umlsec/Internals/umlap/downloads/index.html

Anhang A

Axiome

```
%----------------- Asymmetrical Encryption ------------------

input_formula(enc_equation,axiom,(
! [E1,E2] :
( ( knows(enc(E1, E2))
  & knows(inv(E2)) )
 => knows(E1) ) )).

%----------------- Symmetrical Encryption -------------------

input_formula(symenc_equation,axiom,(
! [E1,E2] :
( ( knows(symenc(E1, E2))
  & knows(E2) )
 => knows(E1) ) )).

%----------------- Signature --------------------------------

input_formula(sign_equation,axiom,(
! [E,K] :
( ( knows(sign(E, inv(K) ) )
  & knows(K) )
 => knows(E) ) )).

%-- Basic Relations on Knowledge where ----------------------
%-- conc, enc, symenc and sign is included ------------------

input_formula(construct_message_1,axiom,(
! [E1,E2] :
( ( knows(E1)
```

```
  & knows(E2) )
=> ( knows(conc(E1, E2))
   & knows(enc(E1, E2))
   & knows(symenc(E1, E2))
   & knows(dec(E1, E2))
   & knows(symdec(E1, E2))
   & knows(ext(E1, E2))
   & knows(sign(E1, E2)) ) ) )).

input_formula(construct_message_2,axiom,(
! [E1,E2] :
( ( knows(conc(E1, E2)) )
=> ( knows(E1)
  & knows(E2) ) ) )).

%-- Basic Relations on Knowledge ----------------------------
%-- where head, tail and hash is included -------------------

input_formula(construct_message_3,axiom,(
! [E] :
( knows(E)
=> ( knows(head(E))
   & knows(tail(E))
   & knows(hash(E)) ) ) )).

%------------ decryption, signature verifikation -----------

input_formula(dec_axiom,axiom,(
! [E,K] :
( equal( dec(enc(E, K), inv(K)), E ) ) )).

input_formula(symdec_axiom,axiom,(
! [E,K] :
( equal( symdec(symenc(E, K), K), E ) ) )).

input_formula(sign_axiom,axiom,(
! [E,K] :
( equal( ext(sign(E, inv(K)), K), E ) ) )).

%------------ head, tail, fst, snd, thd, frth --------------


input_formula(tail_axiom,axiom,(
! [X,Y] :
```

```
( equal( tail(conc(X,Y)), Y ) ) )).

input_formula(fst_axiom,axiom,(
! [X] :
( equal( fst(X), head(X) ) ) )).

input_formula(snd_axiom,axiom,(
! [X] :
( equal( snd(X), head(tail(X)) ) ) )).

input_formula(thd_axiom,axiom,(
! [X] :
( equal( thd(X), head(tail(tail(X))) ) ) )).

input_formula(frth_axiom,axiom,(
! [X] :
( equal( frth(X), head(tail(tail(tail(X)))) ) ) )).

%------------------ mac -----------------------------------

input_formula(symmac_axiom,axiom,(
! [X,Y] :
( (knows(X) & knows(Y)) => knows(mac(X, Y)) ) )).
```

Anhang B

Anhang zum Fallbeispiel

Es werden im Folgenden alle erzeugten TPTP-Dateien angegeben. Die Diagramme mit den erzeugten TPTP-Dateien sowie sämtliche Ausgaben der Theorembeweiser E-SETHEO und SPASS stehen über die Webseite des SequenceAnalyser-Tools [Gil05] zum Download bereit.

B.1 Variante des TLS-Protokolls

B.1.1 TPTP-Datei

Diese TPTP-Datei wurde vom SequenceAnalyser aus dem Diagramm in Abbildung 6.2 erzeugt.

```
%----------------------- Asymmetrical Encryption ------------------------

input_formula(enc_equation,axiom,(
! [E1,E2] :
( ( knows(enc(E1, E2))
  & knows(inv(E2)) )
 => knows(E1) ) )).

%--------------------- Symmetrical Encryption ----------------------------

input_formula(symenc_equation,axiom,(
! [E1,E2] :
( ( knows(symenc(E1, E2))
  & knows(E2) )
 => knows(E1) ) )).

%------------------------- Signature -----------------------------------

input_formula(sign_equation,axiom,(
! [E,K] :
( ( knows(sign(E, inv(K) ) )
  & knows(K) )
 => knows(E) ) )).

%---- Basic Relations on Knowledge where conc, enc, symenc and sign is included ----

input_formula(construct_message_1,axiom,(
! [E1,E2] :
( ( knows(E1)
  & knows(E2) )
=> ( knows(conc(E1, E2))
   & knows(enc(E1, E2))
   & knows(symenc(E1, E2))
   & knows(dec(E1, E2))
   & knows(symdec(E1, E2))
   & knows(ext(E1, E2))
   & knows(sign(E1, E2)) ) ) )).
```

```
input_formula(construct_message_2,axiom,(
! [E1,E2] :
( ( knows(conc(E1, E2)) )
=> ( knows(E1)
  & knows(E2) ) ) )).

%---- Basic Relations on Knowledge where head, tail and hash is included ----

input_formula(construct_message_3,axiom,(
! [E] :
( knows(E)
=> ( knows(head(E))
   & knows(tail(E))
   & knows(hash(E)) ) ) )).

%-------------------------- decryption, signature verifikation -----------------

input_formula(dec_axiom,axiom,(
! [E,K] :
( equal( dec(enc(E, K), inv(K)), E ) ) )).

input_formula(symdec_axiom,axiom,(
! [E,K] :
( equal( symdec(symenc(E, K), K), E ) ) )).

input_formula(sign_axiom,axiom,(
! [E,K] :
( equal( ext(sign(E, inv(K)), K), E ) ) )).

%-------------------------- head, tail, fst, snd, thd, frth -----------------------------------

input_formula(head_axiom,axiom,(
! [X,Y] :
( equal( head(conc(X,Y)), X ) ) )).

input_formula(tail_axiom,axiom,(
! [X,Y] :
( equal( tail(conc(X,Y)), Y ) ) )).

input_formula(fst_axiom,axiom,(
! [X] :
( equal( fst(X), head(X) ) ) )).

input_formula(snd_axiom,axiom,(
! [X] :
( equal( snd(X), head(tail(X)) ) ) )).

input_formula(thd_axiom,axiom,(
! [X] :
( equal( thd(X), head(tail(tail(X))) ) ) )).

input_formula(frth_axiom,axiom,(
! [X] :
( equal( frth(X), head(tail(tail(tail(X)))) ) ) )).

%-------------------------- mac ----------------------------------

input_formula(symmac_axiom,axiom,(
! [X,Y] :
( (knows(X) & knows(Y)) => knows(mac(X, Y)) ) )).

%---------------------- Attackers Initial Knowledge ----------------------

input_formula(previous_knowledge,axiom,(
 knows(k_ca)
 & knows(inv(k_a))
 & knows(k_a)
 )).

%---------------------- Main Protocol Specification --------------------------

input_formula(protocol,axiom,(
 ![Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1] : (
 % C -> Attacker
(
     ( ( true
       & true )
      => knows(conc( n,  conc( k_c,  sign(conc(c, conc(k_c, eol)),inv(k_c)) ) ))
     & ( ( knows(Resp_1) & knows(Resp_2)
         & equal( fst(ext(Resp_2,k_ca)), s )
         & equal( snd(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca)))), n ) )
        => knows(symenc(secret, fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca))))))
       )
```

```
     )
)
 &  % S -> Attacker
(
     ( ( knows(Init_1) & knows(Init_2) & knows(Init_3)
       & equal( snd(ext(Init_3,Init_2)), Init_2 ) )
      => knows(conc( enc(sign(conc(kgen(Init_2), conc(Init_1, eol)),inv(k_s)),Init_2),
                     sign(conc(s, conc(k_s, eol)),inv(k_ca)) ))
     & ( ( knows(Xchd_1)
         & true )
        => true
       )
     )
)
 )  ) ).

%----------------------- Conjecture -------------------------

input_formula(attack,conjecture,(
   knows(secret) )).

% Finished
```

B.2 Korrigierte Version des TLS-Varianten-Protokolls

B.2.1 TPTP-Datei

Diese TPTP-Datei wurde vom SequenceAnalyser aus dem Diagramm in Abbildung 6.4 erzeugt.

```
%----------------------- Asymmetrical Encryption -------------------------

input_formula(enc_equation,axiom,(
! [E1,E2] :
( ( knows(enc(E1, E2))
  & knows(inv(E2)) )
 => knows(E1) ) )).

%--------------------- Symmetrical Encryption ----------------------------

input_formula(symenc_equation,axiom,(
! [E1,E2] :
( ( knows(symenc(E1, E2))
  & knows(E2) )
 => knows(E1) ) )).

%-------------------------- Signature -----------------------------------

input_formula(sign_equation,axiom,(
! [E,K] :
( ( knows(sign(E, inv(K) ) )
  & knows(K) )
 => knows(E) ) )).

%---- Basic Relations on Knowledge where conc, enc, symenc and sign is included ----

input_formula(construct_message_1,axiom,(
! [E1,E2] :
( ( knows(E1)
  & knows(E2) )
=> ( knows(conc(E1, E2))
   & knows(enc(E1, E2))
   & knows(symenc(E1, E2))
   & knows(dec(E1, E2))
   & knows(symdec(E1, E2))
   & knows(ext(E1, E2))
   & knows(sign(E1, E2)) ) ) )).

input_formula(construct_message_2,axiom,(
! [E1,E2] :
( ( knows(conc(E1, E2)) )
=> ( knows(E1)
  & knows(E2) ) ) )).

%---- Basic Relations on Knowledge where head, tail and hash is included ----
```

```
input_formula(construct_message_3,axiom,(
! [E] :
( knows(E)
=> ( knows(head(E))
   & knows(tail(E))
   & knows(hash(E)) ) ) )).

%-------------------------- decryption, signature verifikation -----------------

input_formula(dec_axiom,axiom,(
! [E,K] :
( equal( dec(enc(E, K), inv(K)), E ) ) )).

input_formula(symdec_axiom,axiom,(
! [E,K] :
( equal( symdec(symenc(E, K), K), E ) ) )).

input_formula(sign_axiom,axiom,(
! [E,K] :
( equal( ext(sign(E, inv(K)), K), E ) ) )).

%-------------------------- head, tail, fst, snd, thd, frth ------------------------------------

input_formula(head_axiom,axiom,(
! [X,Y] :
( equal( head(conc(X,Y)), X ) ) )).

input_formula(tail_axiom,axiom,(
! [X,Y] :
( equal( tail(conc(X,Y)), Y ) ) )).

input_formula(fst_axiom,axiom,(
! [X] :
( equal( fst(X), head(X) ) ) )).

input_formula(snd_axiom,axiom,(
! [X] :
( equal( snd(X), head(tail(X)) ) ) )).

input_formula(thd_axiom,axiom,(
! [X] :
( equal( thd(X), head(tail(tail(X))) ) ) )).

input_formula(frth_axiom,axiom,(
! [X] :
( equal( frth(X), head(tail(tail(tail(X)))) ) ) )).

%-------------------------- mac ----------------------------------

input_formula(symmac_axiom,axiom,(
! [X,Y] :
( (knows(X) & knows(Y)) => knows(mac(X, Y)) ) )).

%---------------------- Attackers Initial Knowledge ----------------------

input_formula(previous_knowledge,axiom,(
 knows(k_ca)
 & knows(inv(k_a))
 & knows(k_a)
 )).

%---------------------- Main Protocol Specification ---------------------------

input_formula(protocol,axiom,(
 ![Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1] : (
 % C -> Attacker
(
     ( ( true
       & true )
      => knows(conc( n,  conc( k_c,  sign(conc(c, conc(k_c, eol)),inv(k_c)) ) ))
     & ( ( knows(Resp_1) & knows(Resp_2)
         & equal( fst(ext(Resp_2,k_ca)), s )
         & equal( snd(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca)))), n )
         & equal( thd(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca)))), k_c ) )
       => knows(symenc(secret, fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca))))))
       )
     )
)
 &  % S -> Attacker
(
     ( ( knows(Init_1) & knows(Init_2) & knows(Init_3)
       & equal( snd(ext(Init_3,Init_2)), Init_2 ) )
      => knows(conc( enc(sign(conc(kgen(Init_2), conc(Init_1, conc(Init_2, eol))),inv(k_s)),Init_2),
                     sign(conc(s, conc(k_s, eol)),inv(k_ca)) ))
```

```
     & ( ( knows(Xchd_1)
         & true )
       => true
      )
     )
)
 )  ) ).

%----------------------- Conjecture -------------------------

input_formula(attack,conjecture,(
   knows(secret) )).

% Finished
```