Workshop WASP'04

1st Workshop on Analysing Security Protocols

im Rahmen des
Projekts 18.336 Maschinelles Beweisen

Berndt Farwer
(Herausgeber)

Fachbereich Informatik
Universität Hamburg

Juli 2004

Vorwort

Bei der Konzipierung und Programmierung von Softwaresystemen für verteilte Umgebungen spielt der Nachrichtenaustausch eine wichtige Rolle. Die Vernetzung durch das Internet macht dies in vielen Bereichen möglich, jedoch werden zunehmend auch sensible Daten übertragen, die nicht in die falschen Hände geraten dürfen. Dies betrifft E-Commerce-Anwendungen und den elektronischen Bankverkehr ebenso wie Anwendungen z.B. in der Firmen- und Gemeindeverwaltung.

Für diese Bereiche der Nachrichtenkommunikation sind Methoden erforderlich, die sicherstellen, dass der Kommunikationspartner tatsächlich ist, wer er ausgibt zu sein (Authentizität). Weiterhin müssen die Daten für andere unlesbar übermittelt werden (Verschlüsselung) und ggf. die Anonymität Einzelner gewahrt werden.

Die Komplexität heutiger Systeme erfordert zudem den Nachweis über die korrekte Implementierung. Formale Beweise können aber sehr kompliziert sein, so dass häufig nur ein Teil des Systems als formal korrekt bewiesen werden kann. Hierfür ist eine Unterstützung durch Computer-Tools wünschenswert, was unter den Oberbegriff des *maschinellen Beweisens* fällt. Das Vorgehen ist in der Regel nicht voll automatisch, sondern erfordert eine Interaktion bzw. vorherige manuelle Spezifikation bzgl. der geforderten Systemeigenschaften.

Zentrale Infrastuktur-Elemente der Informatik - wie beispielsweise Computer-Chips, Betriebsysteme, Middle-Ware-Componenten - benötigen ebenfalls eine Absicherung der Korrektheit, die nicht mehr durch systematisches Testen, sondern nur noch durch eine formale Verifikation gewährleistet werden kann. Dies ist schon bei einfachen Beispielen nur mit Rechnerhilfe durchführbar.

Gegenstand des studentischen Projektes mit dem Titel „Maschinelles Beweisen“ war die rechnerunterstützte Spezifikation und Verifikation. Zum Einsatz kamen dabei formale Modelle, wie die algebraische Spezifikation von Datentypen (analog zu Typen der Objektorientierung), die Formalisierung nebenläufiger Systeme durch Petrinetze und die Verifikation durch temporale Logik (CTL oder LTL).

Der vorliegende Band stellt die Projektarbeiten aus dem Projekt im Sommersemester 2004 zusammen. Die Beiträge wurden einem zweistufigen Reviewing-Prozess unterzogen bevor sie in der hier abgedruckten Form vorlagen und auf einem Mini-Workshop am 12./13. Juli 2004 präsentiert wurden.

Berndt Farwer — Hamburg, im Juli 2004

Inhaltsverzeichnis

Anna Fricke
0fricke@informatik.uni-hamburg.de
Fachbereich Informatik
Universität Hamburg

1

Protokolle zur Gewährleistung gegenseitiger Anonymität

Dieses Paper beschäftigt sich mit Protokollen, die gegenseitige Anonymität gewährleisten sollen. Ein zentrales Problem von Peer-to-Peer Systemen ist die Wahrung der Anonymität zwischen Initiator und Responder während des Informationsaustausches. Viele bereits existierende Lösungen erreichen gegenseitige Anonymität in reinen Peer-to-Peer Systemen ohne eine vertrauenswürdige zentrale Kontrollinstanz. Ein bekanntes Beispiel ist das Freenet Protokoll. Zum Vergleich wird das „Mix-Based Protokoll“ vorgestellt, das von der Existenz eines vertrauenswürdigen Indexservers ausgeht. Um Lauffähigkeit und Korrektheit solcher Protokolle zu testen bzw. zu beweisen, benötigt man eine formale Spezifikation. Unter Verwendung der Rewriting Engine Maude wird ein Beispiel einer Spezifikation des „Mix-Based Protokolls“ erläutert. Im zweiten Teil des Papers wird ein weiteres Protokoll die „Dining Cryptographers“ und eine mögliche Spezifikation in Maude betrachtet, die auf dem Modell eines Petrinetzes aufbaut. Diese Spezifikation des Protokolls wird durch Model-Checking verifiziert.

Keywords: Anonymität, Spezfikation, Model Checking, Maude

1.1 Einleitung

Grundsätzlich kann man Anonymität so verstehen, dass man in Erscheinung tritt, ohne seine Identität preis zu geben. Sie ist nicht zwangsläufig auf Personen beschränkt, so können sich z.B. Computersysteme anonym oder identifizierbar an einem anderen System anmelden [Wik].

Bei Aktivitäten im Internet fühlen sich viele Benutzer anonym. Diese Anonymität ist jedoch trügerisch. Grundsätzlich erfährt die Gegenseite bei der Kommunikation die IP-Adresse. Doch auch Cookies oder Browserinformationen können ohne Wissen des Anwenders weitergegeben werden. Mit der IP-Adresse eines Benutzers kann der Anbieter von Internetdiensten die tatsächliche Identität des Benutzers nicht ermitteln, er kann jedoch Hinweise wie den Provider und oft auch noch Land und Region herausfinden. Für die Identität muss eine Anfrage beim Provider erfolgen, dieser besitzt die nötigen Daten. Um die IP-Adresse beim Surfen zu verschleiern, werden oft anonymisierende Proxyserver benutzt. Der Proxybetreiber kennt aber immer noch die IP-Adresse des Nutzers, und kann diese auf Anfrage herausgeben. Um das zu vermeiden bauen bestimmte Tools Ketten von Proxies auf, zwischen denen der Verkehr verschlüsselt wird. Diese Variante ist langsam, aber recht sicher, da nur eine fehlende Zwischenstation die Rekonstruktion unmöglich macht. Will man anonym Daten veröffentlichen oder Dateien tauschen, kommen oft anonyme Peer-to-Peer-Netzwerke zum Zug. Sie funktionieren ähnlich, mit mehreren Zwischenstationen und Verschlüsselung an jedem Pfad.

Grundsätzlich kann man P2P Systeme in zwei Klassen unterteilen: Ein dezentrales P2P System, in dem die Daten ohne ein zentral operierende Einheit geteilt werden und ein hybrides P2P System, in

dem einige Operationen wie das Indizieren der Dokumente zentral gesteuert werden. In einem P2P System können die einzelnen Peers unterschiedliche Rollen einnehmen: Einmal der Publisher, der die Dokumente erstellt, der Initiator, der die Dokumente anfordert und der Responder, der die Daten verwaltet und für Anfragen bereit stellt. Gegenseitige Anonymität bedeutet, dass weder Initiator noch Responder sich gegenseitig identifizieren können oder von anderen Peers identifiziert werden. Ziel ist es, bereits existierende Protokolle zur Gewährleistung gegenseitiger Anonymität vorzustellen. Hier werden zwei Ansätze unterschieden: Der für ein dezentrales P2P System und der für ein hybrides P2P System, bei dem von der Existenz eines vertrauenswürdigen Indexservers ausgegangen wird. Für den ersten Fall wird das Protokoll vom Freenet Projekt betrachtet. Freenet ist eine adaptive P2P Applikation, die von Ian Clarke entwickelt wurde, um die Identität von Publisher, Initiator und Responder zu schützen [CSWH]. Anschliessend wird ein weiteres Protokoll betrachtet, das für ein hybrides P2P System entwickelt wurde. Das „Mix-Based Protokoll“ . Die Spezifikation des Protokolls wurde in Maude entwickelt, da sich die Rewriting Logic Engine besonders gut dafür eignet, die Eigenschaften des Protokolls schon während der Entwicklungphase zu testen [CDE+03]. Um zu veranschaulichen, wie ein Protokoll gegenseitige Anonymität gewährleisten kann, bei dem die entsprechenden Informationen abstrahiert werden, wird anschließend beispielhaft das "Dining CryptographersProtokoll diskutiert. Zunächst wird das Protokoll in einem Petrinetz modelliert und dann in Maude spezifiziert. Durch Einsatz des sogenannten Model-Checkings wird das Protokoll verifiziert.

1.2 Anonymität in dezentralen P2P Systemen

In einem P2P-Netzwerk ist jeder angeschlossene Computer Server und Client zugleich. Jeder Peer bestimmt selbst welche Resourcen er dem P2P-Netz zur Verfügung stellt.

Anonymität betrifft hier den Schutz der einzelnen Agenten hinsichtlich bestimmter Ereignisse und dem Austausch von Nachrichten. Ein Protokoll zur Gewährleistung gegenseitiger Anonymität sollte daher einigen Designkriterien entsprechen. Ziel ist nicht den Inhalt der Nachricht zu schützen, sondern nur ihre Verbindung zu den einzelnen Peers. Es ist auch nicht notwendig, dass die Identität zweier Peers einander beim direkten Nachrichtenaustausch verborgen bleibt.

Wir nehmen an, A sendet eine Nachricht M an B über C. C kann hier ein Knoten im Pfad von A nach B sein oder ein Eindringling, der die Leitung abhört. A darf zu keinem Zeitpunkt wissen, dass M an B gesendet wird und B darf auf der anderen Seite auch nicht wissen, dass M von A kommt. Liest C M, so darf C nicht wissen, dass M von A erzeugt wurde und an B gesendet wird.

Das Problem der gegenseitigen Anonymität in dezentralen P2P Systemen gab Anstoß für die Entwicklung einiger Applikationen, die diese wahren sollten. Einige bekannte Beispiele sind hier Gnutella, Kazaa oder Freenet. Allerdings ergeben sich bei dezentralen P2P Systemen zusätzliche Probleme, wie Sabotageresistenz, keine optimale Lastverteilung oder unkontrollierte Wucherungen des Netzes. Im Folgenden werden wir aber nicht weiter auf diese Probleme eingehen.

1.3 Freenet

Freenet [CSWH] wurde im Jahre 1999 von Ian Clark als Student an der Universität von Edinburgh entwickelt und ist in Java implementiert. Ziel war es, ein verteiltes, zensurresistentes Informations Speicher- und Anfragesystem zu entwickeln, das ermöglicht, Daten anonym zu publizieren, zu speichern und abzufragen. Man kann fünf entscheidende Designkriterien erkennen:

- Dezentralisiertheit aller Netzwerkfunktionen

- Anonymität für Autoren und Konsumenten von Informationen

- Dementierbarkeit des Besitzes von Information

- Resistenz vor Dritten, die den Zugang zu Informationen verhindern wollen

- Effizientes dynamisches Speichern und Routen der Informationen

Bei Freenet werden die Daten redundant und verteilt gespeichert, so dass es schwer ist, Informationen zu entfernen oder ihren Herausgeber zu identifizieren. Jedes Mal, wenn eine Datei auf eine Anfrage versendet wird, wird diese auf jeder Zwischenstation in einem Cache gespeichert. Wenn das Maximum des Caches erreicht ist, werden die Dokumente mit den wenigsten Anforderungen gelöscht. Gleichzeitig werden die Transfers durch mehrere Hosts geroutet, so dass eine weitere Replikation erreicht wird. Alle im System vorhandenen Daten werden mit so genannten Keys gespeichert. Nur wer diesen Klartextkey kennt, kann die Informationen auch abfragen.

1.3.1 Die Architektur

Freenet ist ein adaptives P2P Netzwerk. Das bedeutet, dass sich die einzelnen Knoten mit der Zeit an die Struktur des Netzes anpassen. Sie lernen, wo sich Informationen befinden und passen ihre Routingtabellen dem entsprechend an.

Jede Anfrage wird an den unmittelbaren Nachbarn geschickt, in der Hoffnung, dass sich die angeforderten Informationen in seinem Speicher befinden. Wenn er das entsprechende Dokument nicht in seinem Cache findet, sendet er die Anfrage an den lexikographisch[1] nächsten Host weiter. Durch dieses Routing bilden sich lexikographische Cluster, die eine Selbsoptimierung bewirken sollen. Die Anfrage wird nun solange weiter geschickt bis entweder das „hops-to-live Limit “erreicht ist, kein weiterer Host mehr auf der Route ist oder das Dokument gefunden wurde. Das „hops-to-live Limit“gibt an, wie viele Knoten maximal traversiert werden können. Zusätzlich wartet jeder Knoten nur eine bestimmte Zeit lang auf eine Antwort auf sein Request. Ist dieser Timeout verstrichen, geht er davon aus, dass die Informationen nicht gefunden wurden. Ist das entsprechende Dokument gefunden, wird es auf dem selben Pfad zurückgeschickt. Jeder Knoten kennt aber nur seine unmittelbaren Nachbarn und kann daher nicht erkennen, wer die Information angefordert hat und von wem sie kommt.

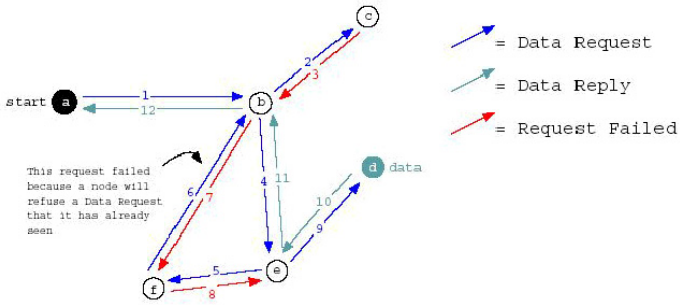


Abbildung 1.1: Eine Anfrage in Freenet

Dateien werden in Freenet über binäre Schlüssel identifiziert, die durch Anwendung einer Hash-Funktion erzeugt werden. Momentan wird die 160-bit SHA-1 Funktion verwendet. Es existieren drei wichtige Schlüsseltypen:

KSK (keyword-signed-key) : Er wird aus einem kurzen beschreibenden Textstring abgeleitet, welcher der Nutzer beim Einfügen der Datei festlegt. Aus diesem String wird ein public/private Schlüsselpaar generiert. Der öffentliche Teil liefert gehashed den File-Key, mit dem die Datei verschlüsselt wird. Der

[1]Jeder Knoten verwaltet eine Routing-Tabelle, in der er die aktuelle IP Adresse seiner unmittelbaren Nachbarn und die File-Keys der bekannten Dateien im Speicher seines Nachbarn verwaltet. Die lexikographische Ordnung richtet sich nach den File-Keys

private Teil dient zur Signierung zwecks Integritätscheck. Zum Wiederfinden der Datei wird nur der Beschreibungsstring veröffentlicht. Ein grosses Problem ist hier die gleiche Namensgebung.

SSK (signed-subspace-key): Sie erlauben dem Autor seine Dateien unter dem selben Schlüssel zu speichern. Er entsteht aus dem oben genannten öffentlichen Teil und dem gehashten Beschreibungsstring.

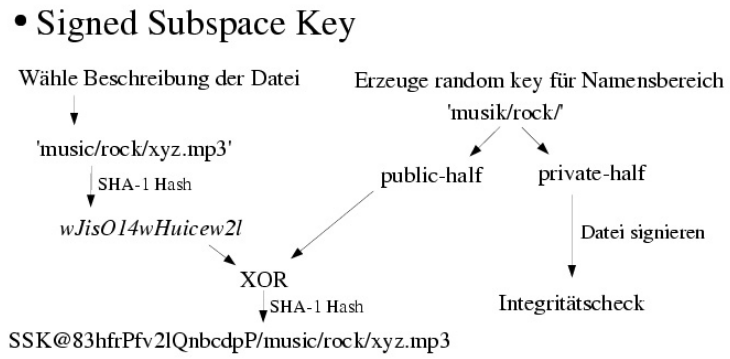


Abbildung 1.2: Erzeugung des Signed Subspace Keys

CHK (content-hash-key) : Es wird ein Hash-Wert über dem Inhalt der Datei gebildet. Er dient der Implementierung von file updating und splitting. Die Dateien werden ausserdem noch mit einem zufällig generierten Key verschlüsselt. Er wird dann zusammen mit dem decryption key veröffentlicht.

Da die Dokumente in den Caches verschlüsselt vorliegen, kann keiner der Besitzer den Namen oder Inhalt des Dokumentes erkennen. Nur wenn man ein Dokument aus seinem eigenem Cache anfordert, weiß man ob diese auf dem eigenen Host überhaupt existiert. Die Keys selber liegen im lokalen Index nur in der Form von Prüfsummen vor, die nicht zurückverwandelt werden können.

Die lexikographische Ordnung der Schlüssel lässt nicht automatisch auf die semantische Verwandschaft der originalen Beschreibungsstrings schliessen. Der Routing-Algorithmus basiert auf der Lokalisation von Keys und nicht der Dokumente selbst. Dadurch werden Dokumente mit thematisch ähnlichem Inhalt nicht auf einem einzigen Knoten gespeichert.

1.4 Anonymität mit einem vertrauenswürdigen Dritten

Wir nehmen bei dem folgenden Protokoll an, dass ein vertrauenswürdiger Indexserver in einem dezentralen P2P System existiert . Dieser Server verwaltet die veröffentlichen FileIds und die Knoten, auf denen die entsprechenden Dateien gespeichert sind. Jeder Peer sendet einen Index der Dateien, die er veröffentlichen will an den Indexserver in periodischen Abständen oder wenn er eine Änderung an den von ihm veröffentlichten Dateien vornehmen will. In Folgendem steht I für den Initiator, R für Responder und S für den Indexserver. Die einzelnen Peers werden durch P_i(i=1,2,3...) repräsentiert. Des weiteren benutzen wir $X->Y$: M für die Nachricht M, die X an Y schickt. K_x ist der public key von X und (M)K ist die mit K verschlüsselte Nachricht M.

1.4.1 Mix-Based Protokoll

1.: Der Initiator sendet eine Anfrage auf ein Dokument an den Indexserver, die er mit dem öffentlichen Schlüssel des Indexservers verschlüsselt.

```
I -> S : (file_Id)Ks
```

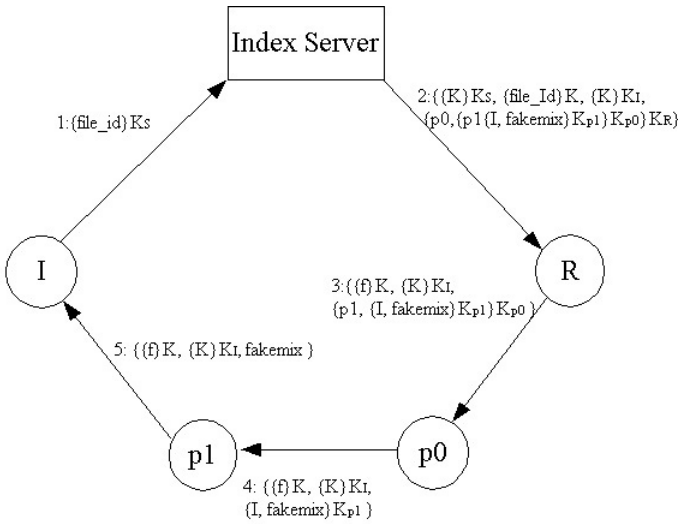


Abbildung 1.3: Mix-Based Protokoll

2.: Der Indexserver sucht in seinen Einträgen, wo das entsprechende Dokument gespeichert ist und erzeugt einen symmetrischen Schlüssel K und eine zufällige Liste von Peers, die den Pfad angeben, den das Dokument nehmen soll. Der Pfad hat folgende Form:

$$\text{path:}(\mathtt{P}_0,\mathtt{P}_1,\ldots(\mathtt{I},\mathtt{fakemix})\mathtt{K}_p I)\mathtt{K}_p k\ldots)\mathtt{K}_p 1)\mathtt{K}_p 0)\mathtt{K}_R \quad .$$

Der fakemix in der Mitte soll einen weiterlaufenden Pfad vortäuschen. Nur I kann erkennen, dass es sich nicht um weitere Knoten im Pfad handelt, so dass Pk nicht weiß, dass er der letzte Knoten vor dem Initiator ist. Der Indexserver sendet dann eine Nachricht an R. Sie enthält die FileID (file_Id) verschlüsselt mit dem symmetrischen Schlüssel K und zweimal den Schlüssel K, einmal verschlüsselt mit dem public Key von R KR und einmal mit dem von I. Ausserdem enthält sie noch den Pfad, den er soeben generiert hat.

S -> R : (K)K_R, (K)K_I, (file_Id)K, path

3.: R entschlüsselt nun den symmetrischen Schlüssel K, indem er seinen private Key benutzt. Mit K kann er dann die FileId entschlüsseln, um die angeforderte Datei f zu identifizieren, die er dann wieder mit K verschlüsselt. Nun entschlüsselt er noch den Pfad mit seinem private Key und verschickt diesen an den im Pfad nächsten Knoten zusammen mit der verschlüsselten Datei und dem noch verschlüsselten K.

R -> $\mathtt{P}_0$: (K)K_I, (f)K, path

4.: Pi entschlüsselt den Pfad mit seinem private Key und sendet die Nachricht an den im Pfad nächsten Knoten weiter.

$\mathtt{P}_i$ -> $\mathtt{P}_{i+1}$: $(\mathtt{K})_I$, (f)K, rest of path

5.: I erhält den Schlüssel K mit seinem private Key und kann dann mit K f entschlüsseln.

Dieses Protokoll gewährt absolute Anonymität [XXZ03b]. Weder Initiator, noch Responder oder einer der Peers auf dem Pfad können die Identität der anderen erkennen. Allerdings muss man einen grossen Overhead in Kauf nehmen, da der Pfad mehrmals mit einem public Key verschlüsselt wird. Hinzu kommt, dass es passieren kann, dass ein Knoten im Pfad oder der Indexserver ausfällt. Hier

bietet das Protokoll keine Alternativen, um dieses Problem zu umgehen.

1.5 Dining Cryptographers

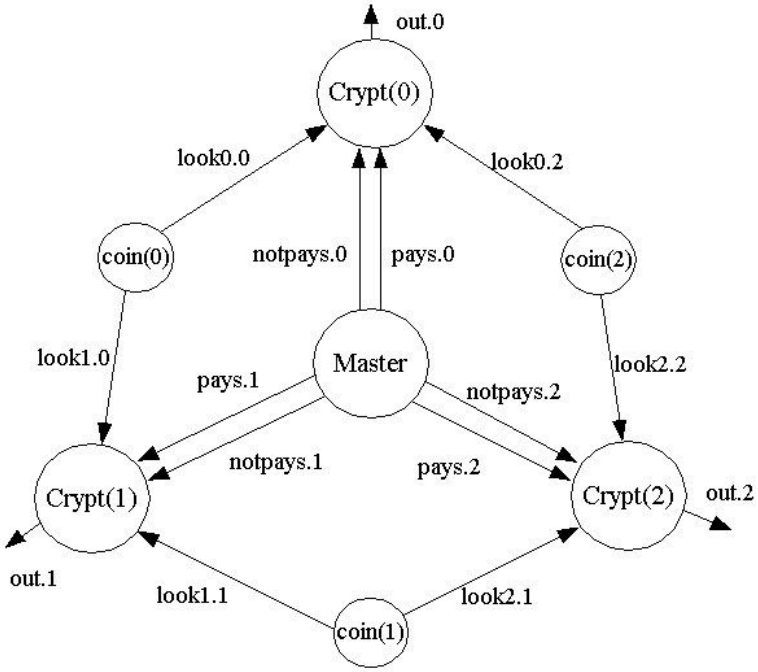


Abbildung 1.4: Dining Cryptographers

Wir haben gesehen, dass Anonymität, soweit es dabei um den Schutz der Identität der Agenten in einem vernetzten System handelt, im Zusammenhang mit den einzelnen Ereignissen und den versendeten Nachrichten steht. Zusätzlich ist es einem Beobachter im Allgemeinen nicht möglich, Einsicht in jede gesendete Nachricht zu haben und gleichzeitig jedes Ereignis zu kennen. Er hat eher einen beschränkten Zugang zu Informationen. So relativiert sich die Anonymität mit der Menge an Informationen über das System, zu denen der Beobachter Zugang hat. Daher ist es oft zweckmässig, die Informationen, die es zu schützen gilt, zu abstrahieren.

Als Beispiel betrachten wir hierfür das Protokoll "Dining Cryptographers "[RS01]. Man geht von der Situation aus, dass drei Cryptographen bei einem abendlichen Geschäftsessen sitzen. Am Ende des Essens wird nun jeder der drei seperat von ihrem gemeinsamen Chef informiert, ob er das Essen bezahlen soll oder nicht. Dabei gibt es einmal die Möglichkeit, dass einer der Cryptographen zahlt oder der Chef übernimmt die Rechnung.

Nun möchten die Cryptographen aber, nachdem jeder informiert wurde, wissen, ob der Chef zahlt oder einer von ihnen. Gesetz dem Fall, dass einer von ihnen zahlt, möchte der aber nicht, dass seine Identität bekannt wird. Also wirft jeder der drei eine Münze und zeigt den Wurf seinem rechten Nachbarn. So kennt jeder seinen eigenen Wurf und den von seinem linken Nachbarn. Dann sagt jeder der drei, ob die beiden Würfe übereinstimmen oder nicht. Jeder der nicht zahlen muss ist angewiesen die Wahrheit zu sagen und der der Zahlen muss, soll das Gegenteil sagen.

Ist die Anzahl der Aussagen, dass die Münzen **nicht** übereinstimmen gerade, so zahlt der Chef. Sind sie ungerade, so können sie sich sicher sein, dass einer von ihnen zahlt. Aber die beiden Cryptographen, die nicht zahlen müssen, werden nicht erkennen können, welcher der anderen beiden der ist, der zahlt.

1.6 Spezifikation der Dining Cryptographers

Rewriting Logic ist ein ausführbarer Spezifikations-Formalismus, der einen semantischen Rahmen bietet, verschiedene Modelle, die Nebenläufigkeit beschreiben, auszudrücken. Um dieses näher zu erläutern, wollen wir das “Dining Cryptographers Protokoll “ zunächst in in einem Petrinetz modellieren, um dieses dann anschliessend in eine Maude Spezifikation zu überführen.

Ein Petrinetz für die Dining Cryptographers sähe zum Beispiel wie in Abbildung 4 aus. Die Schaltfunktionen der einzelnen Transitionen werden im einzelnen an der entsprechenden Passage der Spezifikation in Maude erläutert. Als Sorten benötigen wir zunächst *Coin*, *Crypt*, *Marking*, *Token* und *Coi-*

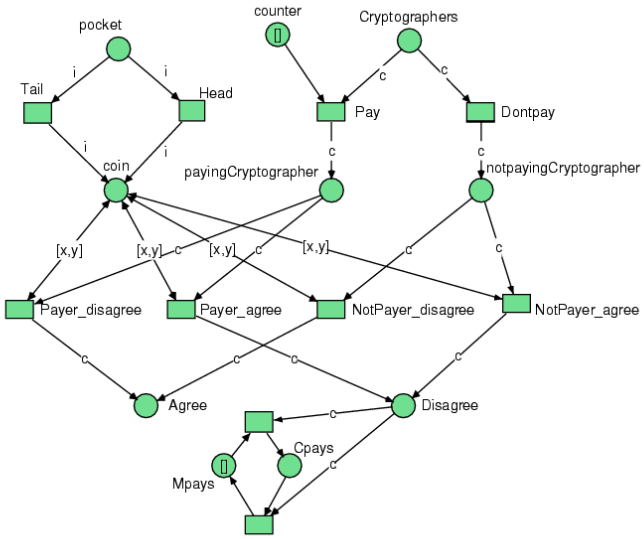


Abbildung 1.5: Das “Dining Cryptographers Protokoll“als Petrinetz

nId, wobei *Token* eine Subsorte von *Marking* ist.

Durch die Operationen *m-nil* und – wird das Schalten der Transitionen eines Petrinetzes spezifiziert. Für jede Münze und jeden Cryptographen steht eine Konstante, die in der Anfangsmarkierung in den Plätzen *pocket* bzw *Cryptographers* als Token liegt.

```
ops c0 c1 c2 : -> Crypt .
ops coin0 coin1 coin2 : -> CoinId .

op pocket : CoinId -> Token .
op Cryptographer : Crypt -> Token .
```

Durch das Schalten der Transitionen *Pay* und *Dontpay* wird der Entscheidungsprozess, ob ein Cryptograph zahlen soll oder nicht, simuliert. Der Platz *counter* stellt sicher, dass die Transition *Pay* nur einmal schalten kann, also höchstens eine Marke auf dem Platz *payingCryptographer* liegen kann.

```
rl [Pay] :
    Cryptographers(c) counter => payingCryptographer(c) .

    rl [Dontpay] :
    Cryptographers(c) => notpayingCryptographer(c) .
```

Die Tansitionen *Tail* und *Head* stehen für den Münzwurf. sie nehmen eine CoinId und wenden die Operation *head* oder *tail* an und legen einen Coin als Token in die Stelle *coin*.

```
rl [Head] :
        pocket(i) => coin(head(i)) .
rl [Tail] :
        pocket(i) => coin(tail(i)) .

op head : CoinId -> Coin . op tail : CoinId -> Coin .
```

Die Transitionen *Payer_disagree* und*Payer_agree* nehmen sich einen Token vom Typ Crypt aus dem Platz *payingCryptographer* und zwei Coins aus der Stelle *coin*. Der eine Coin muss seiner eigenen Münze entsprechen und der andere dem seines linken Nachbarns. Diese Zugehörigkeit wird durch die Operationen *rightCoin* und *leftCoin* bestimmt. Um dies zu prüfen, muss man erst aus dem Token,der als Coin vorliegt, die CoinId auslesen. Dies geschieht über die Operation *id*.

```
    op rightCoin : Crypt -> CoinId .
    eq rightCoin(c0) = coin0 . eq rightCoin(c1) = coin1 .
    eq rightCoin(c2) = coin2 .

    op leftCoin : Crypt -> CoinId .
    eq leftCoin(c0) = coin2 . eq leftCoin(c1) = coin0 .
    eq leftCoin(c2) = coin1 .

    op id : Coin -> CoinId .
    eq id(head(i)) = i . eq id(tail(i)) = i .
```

Die Transition *Payer_disagree* schaltet nur unter der Bedingung, dass die beiden Coins den gleichen Wurf anzeigen. Dies wird über die Operationen *isHead* und *isTail* geprüft, die einen Boolean zurückgeben. Die Transition *Payer_agree* schaltet nur, wenn die beiden Münzen einen unterschiedlichen Wurf anzeigen.

```
op isHead : Coin -> Bool .
eq isHead(head(i)) = true . eq isHead(tail(i)) = false .
op isTail : Coin -> Bool .
eq isTail(tail(i)) = true . eq isTail(head(i)) = false .

crl [Payer_disagree] :
payingCryptographer(c) coin(x) coin(y) => Disagree(c) coin(x) coin(y)
if id(x) == leftCoin(c) and id(y) == rightCoin(c) and isHead(x) == isHead(y).

crl [Payer_agree] : payingCryptographer(c) coin(x) coin(y) =>
Agree(c) coin(x) coin(y) if id(x) == leftCoin(c) and id(y) ==
rightCoin(c) and isHead(x) =/= isHead(y).
```

Die Transitionen *Notpayer_disagree* und *Notpayer_agree* schalten genau entgegengesetzt der oben genannten Bedingungen. Die Cryptographen liegen nun in den Plätzen *Agree* und *Disagree*. Wir wissen aus der Protokollbeschreibung, dass wenn eine ungerade Anzahl der Cryptographen sagt, dass der Wurf der Münzen nicht übereinstimmt, was den Marken in der Stelle *Disagree* entsprechen würde, sie wissen, dass einer von ihnen die Rechnung übernimmt. Folglich muss man nun nur die Anzahl der Marken in der Stelle *Disagree* zählen. Zu diesem Zweck schlaten die Transitionen *odd* und *even*.

```
    rl [odd] :
            Disagree(c) Masterpays => Cryptpays .
    rl [even] :
            Disagree(c) Cryptpays => Masterpays .
```

1.6.1 Verifikation durch Model-Checking und Erweiterbarkeit

Vorweg muss an dieser Stelle gesagt werden, dass sich die Spezifikation nur dafr eignet, das "Dining Cryptographers Protokoll"auf seine Korrektheit hin zu verifizieren, nicht aber auf die Gewährleistung der Anonymität. Hierfür müsste zusätzlich das Wissen der einzelnen Cryptographen über die anderen modelliert werden.

Da dieses Protokoll nur endlich viele Zustände hat, bietet sich zur Verifikation der Spezifikation das Model-Checking an. Maude bietet hier die Semantische Umgebung, das Protokoll als Kripke-Struktur zu betrachten und mittels Temporallogischer Formeln, die Korrektheit zu bestätigen oder zu widerlegen [CDE+03]. Des Weiteren eignet sich diese Methode, um zu testen, ob das Protokoll auf die Anzahl von drei Cryptographen beschränkt ist oder erweiterbar ist. Wir definieren zunächst die Zustände des Petrinetzes für den eingebundenen Model-Checker.

```
inc MODEL_CHECKER .

subsort Marking < State . var m : Marking .
```

Dann definieren wir die Eigenschaften, auf die der Model-Checker prüfen soll. Die anschliessenden Gleichungen geben an, wann diese Eigenschaft wahr ist.

```
ops isPaying isNotpaying : -> Prop .

eq ((m payingCryptographer(c) |= isPaying) = true .
eq ((m notpayingCryptographer(c) |= isNotpaying) = true .

ops Mpays Cpays : -> Prop .

eq ((m Masterpays) |= Mpays) = true .
eq ((m Cryptpays) |= Cpays) = true .
```

Über die Operation *phi0* prüfen wir, ob auf allen Pfaden immer nicht gilt: Der Chef der Firma zahlt und einer der Cryptographen sollte auch bezahlen.

```
op phi0 : -> Prop .
eq phi0 = [] ~(isPaying /\ Mpays) .
```

Um zu testen, ob das "Dining Cryptographers Protokoll"auch bei mehr als drei Cryptographen funktioniert, muss man nur die entsprechenden Stellen in der Spezifikation erweitern.

```
ops c0 c1 c2 c3 \dots : -> Crypt .
ops coin0 coin1 coin2 coin3 \dots : -> CoinId .

eq rightCoin(c0) = coin0 . eq rightCoin(c1) = coin1 .
eq rightCoin(c2) = coin2 . eq rightCoin(c3) = coin3 .  ...

eq leftCoin(c0) = coin\textbf{3} . eq leftCoin(c1) = coin0 .
eq leftCoin(c2) = coin1 leftCoin(c3) = coin\textbf{2} .  ...
```

Prüft man nun auf die oben genannte LTL-Formel, sieht man, dass sie bei jeder gewählten Anzahl von Cryptographen wahr ist. Also bleibt die Anzahl von Marken in der Stelle *Disagree* immer ungerade, wenn einer der Cryptographen zahlt. Den Beweis, ob die Zahl der Cryptographen beliebig d.h. unendlich erweiterbar ist, bleibe ich an dieser Stelle schuldig. Um diesen Beweis zu erbringen, benötigt man einen Formalismus, der anders als Model Checking ein System mit unendlichem Zustandsraum verifizieren kann.

1.7 Spezifikation des Mix-Based Protokolls

Anders als bei der Spezifikation des "Dining Cryptographers Protokolls", die auf dem Modell eines Petrinetzes aufbaut, verwenden wir beim Mix-Based Protokoll das in Maude2.1 enthaltene fullmaude Package. Dies ermöglicht eine objektorientierte Programmierung in Maude. Um dies zu demonstrieren, modellieren wir Initiator und Responder als Objekte der Klasse *Agent*.

Die Sorten für die Spezifikation stehen in folgender Hierarchie:

```
protecting QID .
protecting INT .

sorts Key Principal Server Person FileId Field FieldSet NULL .

subsort Qid < Oid .
subsorts Oid < Principal Person.

subsorts Key FileId Person < Field .
subsort NULL < Key FileId Field FieldSet .
subsorts Principal Server < Person .
subsort Field < FieldSet .
```

Principal und *Person* sind hier ObjectIdentifier. Für das Ver- und entschlüsseln der Nachrichten definieren wir die Operation *ed*.

```
op ed : Key FieldSet -> FieldSet .
var k : Key .
var S : FieldSet .
eq ed(k,ed(k,S)) = S .
```

Um die Spezifikation zu vereinfachen, wird nicht zwischen asymmetrischer und symmetrischer Verschlüsselung unterschieden. Der *SecretKey* ist der symmetrische Schlüssel, den der Indexserver für den Datenaustausch zwischen Initiator und Responder generiert.

```
op PublicKey : Principal -> Key .
op ServerKey : Server -> Key .
op SecretKey : -> Key .
```

Die Operation *from_to_send* verschickt eine Nachricht von einer Person an eine andere. Auf der Nachricht sind einige Operationen definiert:

```
op add : FieldSet FieldSet -> FieldSet  [assoc id: NAF ] .
op rest : FieldSet -> FieldSet .
op first : FieldSet -> FieldSet .
vars fs1 fs2 : FieldSet .
eq rest(add(fs1,fs2)) = fs2 .
eq first(add(fs1,fs2)) = fs1 .
```

Die Klasse *Agent* besitzt die Attribute *Akey*: Key und *reqF*: FieldSet.

Die Regel [Begin] definiert, wie der Initiator eine Nachricht an den trusted Indexserver sendet, die die FileId der gewünschten Datei enthält, verschlüsselt mit dem Public Key des Servers.

```
crl [Begin] :
       < I : Agent |  reqF : requestFile(x,R) , Akey : PublicKey(I) >
       =>
       < I : Agent |  reqF : NAF , Akey : PublicKey(I) >
       from(I)to(IS)send(ed(ServerKey(IS),requestFile(x,R)))
       if I =/= NAF .
```

Die Regel [ISRecSnd] definiert, wie der Server die Nachricht vom Initiator erhält und eine Nachricht an den Responder verschickt. Das Generieren des Fakemixes und des SecretKeys findet ausserhalb des Protokolls statt und wird hier nicht modelliert.

```
rl [ISRecSnd] :
    from(I)to(IS)send(ed(ServerKey(IS),requestFile(x,R)))
    =>
    from(IS)to(R)send(add(ed(SecretKey,requestFile(x,R)),
    ed(PublicKey(R),SecretKey),
    ed(PublicKey(I),SecretKey),
    ed(PublicKey(R), add(R,(ed(PublicKey(P1), add(P1,(ed(PublicKey(P2),
    add(P2,(ed(PublicKey(I),(add(I,Fakemix)))))))))) ) )  .
```

In der Regel [RRecSnd] definieren wir, wie der Responder die Nachricht des Indexservers erhält und die FileId entschlüsselt und die entsprechende Datei mit dem SecretKey verschlüsselt an den nächsten Knoten im Pfad verschickt.

```
rl [RRecSnd] :
    < R : Agent | Akey : PublicKey(R) >
    from(IS)to(R)send(add(ed(SecretKey,requestFile(x,R)) ,
    ed(PublicKey(R),SecretKey),
    ed(PublicKey(I),SecretKey),P))
    =>
    < R : Agent | Akey : PublicKey(R) >
    from(R)to(P1)send(add(ed(ed(PublicKey(R),ed(PublicKey(R),SecretKey)),getFile),
    ed(PublicKey(I),SecretKey),
    rest(ed(PublicKey(R),P)) )) .
```

Die folgende Regel zeigt, wie ein Knoten auf dem Pfad die Nachricht enmpfängt, entschlüssselt und weiter schickt.

```
rl [P1RecSnd] :
    from(R)to(P1)send(add(ed(SecretKey,getFile),
    ed(PublicKey(I),SecretKey),
    S))
    =>
    from(P1)to(P2)send(add(ed(SecretKey,getFile),
    ed(PublicKey(I),SecretKey),
    rest(ed(PublicKey(P1),S)) )) .
```

In der Regel [IRecFile] empfängt der Initiator die Nachricht vom letzten Knoten im Pfad und entschlüsselt erst den SecretKey mit seinem Public Key und kann dann damit die Datei entschlüsseln.

```
rl[IRecFile] :
     < I : Agent |  reqF : NAF  , Akey : PublicKey(I) >
    from(P2)to(I)send(add(ed(SecretKey,getFile),
    ed(PublicKey(I),SecretKey),
    ed(PublicKey(I),(add(I,Fakemix))) ))
    =>
    < I : Agent |  reqF : ed( ed(PublicKey(I),ed(PublicKey(I),SecretKey)),
    ed(SecretKey,getFile)) , Akey : PublicKey(I) > .
```

1.8 Fazit

Wir haben gezeigt, dass gegenseitige Anonymität sowohl in zentralen als auch in dezentralen P2P Systemen möglich ist. Wie bei allen Protokollen, die sicherheitskritische Bereiche betreffen, ist es von

grosser Bedeutung die Protokolle zu spezifizieren und zu verifizieren. Hier haben wir die Rewriting Logic Engine Maude kennengelernt, die es möglich macht, die Protokolle schon während der Entwicklungsphase zu testen. Ausserdem bietet das Modul Model-Checker.maude die Möglichkeit, die Korrektheit des Sytems bezüglich Temporallogischer Formeln zu beweisen. Die moralischen Bedenken, ein zensurresistentes System zur Speicherung und Anfrage von Informationen zu entwickeln, haben wir bei unseren Überlegungen vernachlässigt und überlassen es dem Auge des Betrachters, zu entscheiden, wie weit die Anonymität beim Publizieren von Informationen in vernetzten Systemen gewährleistet werden sollte.

Literaturverzeichnis

[CDE+03] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *The Maude 2.0 Manual*, June 2003.

[CSWH] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system.

[RS01] P.Y.A. Ryan and S. A. Schneider. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[Wik] Wikipedia. *Wikipedia.org/wiki/Anonymität*.

[XXZ03a] Li Xiao, Zhichen Xu, and Xiaodong Zhang. Low-cost and reliable mutual anonymity protocols in peer-to-peer networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):829–840, 2003.

[XXZ03b] Li Xiao, Zhichen Xu, and Xiaodong Zhang. Mutual anonymity protocols for hybrid peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, (ICDCS'2003), Providence, Rhode Island*, 2003.

Daniel Schradick
1schradi@informatik.uni-hamburg.de
Fachbereich Informatik
Universität Hamburg

2

Ein Vergleich zwischen Model-Checking und Theorem-Proving

In dieser Arbeit werden die zwei Hauptzweige der Verifikation, Model-Checking und Theorem-Proving, vorgestellt. Beide Methoden werden erklärt und verglichen. Überdies wird jeweils ein Stellvertreter genauer beschrieben. Für Model-Checking wurde das Tool SPIN gewählt, während als Theorembeweiser der Beweisassistent Coq Verwendung findet. Nach einer kurzen Einführung in die theoretischen Grundlagen dieser Werkzeuge wird je ein Beispiel für die Beweisführung erbracht. Zum Schluss erfolgt eine Bewertung beider Techniken bezüglich ihrer Eignung in der Softwareverifikation.

Keywords: Model-Checking, Theorem-Proving, Verifikation, Spin, Coq

2.1 Einleitung

Viele Ansätze und Methoden wurden entwickelt, um die Sicherheit und Zuverlässigkeit von Computersystemen zu garantieren. Ein vielversprechender Ansatz ist es, formale Methoden in allen Phasen des Softwareentwicklungsprozesses zu verwenden. Sie setzen Techniken aus mathematischer Logik und diskreter Mathematik ein, um Systeme zu beschreiben, spezifizieren, konstruieren und verifizieren. Ihre Verwendung ermöglicht es, das Verhalten unter bestimmten Annahmen sowie die Eigenschaften eines Systems explizit zu beschreiben. Dies ermöglicht, mittels der zugrundegelegten Logik über das System mit Inferenzregeln zu schließen und dadurch Eigenschaften des Systems zu beweisen.

Der Nachteil der formalen Methoden liegt darin, dass sie zum Teil schwer anzuwenden und sehr zeitintensiv sind. Dies betrifft zum einen die genauen mathematischen Beschreibungen und Spezifikationen des Systems, aber vor allem die Beweisführung, auf deren verschiedene Arten in diesem Bericht genauer eingegangen werden soll.

Es gibt zwei Hauptzweige in der Verifikation, welche durch die Art des Vorgehens bei der Beweisführung gekennzeichnet sind die man mit Theorem-Proving und Model-Checking bezeichnet.

2.2 Theorem-Proving

Beim Theorem-Proving, auch Automated Deduction genannt, wird das System und die zu überprüfende Eigenschaft jeweils in eine Formel einer vorgegebenen Logik transformiert. Die Aussage, dass das System die Eigenschaft besitzt, wird hier dadurch formalisiert, dass die Systemformel ϕ die Formel f der Eigenschaft implizieren muss.

$$\phi \Longrightarrow f$$

Programme, die versuchen, diese Implikation zu zeigen, bezeichnet man als Theorem-Beweiser. Sie sind Systeme, welche zu einem gegebenen Kalkül und einer Formel versuchen, durch wiederholtes Anwenden von Inferenzregeln des Kalküls einen Beweis zu finden. Es gibt viele Variationen, die z.B. die Erweiterbarkeit auf spezielle Logiken und den Grad der Automatisierung betreffen. Generell aber lassen sich diese Beweiser in die Klassen der Interaktiven und Automatischen Theorem-Beweiser einteilen.

2.2.1 Interaktive Theorem-Beweiser

Hier werden die Operationen und Inferenzregeln auf eine Formel manuell angewendet. Dies geschieht durch die Ausführung von Kommandos oder Kommando-Skripten, die man Taktiken nennt. Somit wird der Beweis Schritt für Schritt von Hand erstellt.

Der Vorteil dieses Verfahren ist die große Ausdrucksmächtigkeit und Flexibilität. Es ist möglich, Theorem-Beweiser für Logik höherer Ordnung, getypte Logik oder Logiken, welche auf spezielle Problemklassen zugeschnitten sind, zu erstellen. Dadurch wird es möglich, Beweise auf relativ natürliche und problemorientierte Weise aufzuschreiben. Zudem können komplexere Teile des Beweises, wie Induktion, direkt im Framework der zu Grunde liegenden Logik ausgedrückt werden. Folglich ist es machbar, relativ komplexe Beweise mit Interaktiven Theorem-Beweisern zu erstellen. Durch Hinzufügen von speziellen Taktik- Bibliotheken kann der Beweiser sogar auf bestimmte Anwendungsdomänen optimiert werden, was zu einem höheren Automatisierungsgrad führt. Nach Beendigung des Beweises besteht die Möglichkeit, das Theorem als Lemma in eine Datenbank zu speichern und den kompletten Beweis noch einmal durchzulesen.

Interaktive Theorem-Beweiser haben allerdings auch einige Nachteile. Viele dieser Beweiser sind zu interaktiv - selbst die kleinsten Schritte im Beweis, wie z.B. bei der Induktionsverankerung eine Variable auf 0 zu setzen, müssen von Hand durchgeführt werden. Bei fehlerhaften Spezifikationen oder Systemen wird zudem keine Rückmeldung darüber gegeben, an welcher Stelle ein Fehler vorliegt, sondern dies drückt sich lediglich dadurch aus, dass das Anwenden der Regeln nicht zur gewünschten Implikation führt. Es ist somit nur schwer ersichtlich, ob ein Fehler vorliegt oder der Beweis anders geführt werden muss. Folglich können große und komplizierte Beweise wochen- oder monatelang dauern. Zudem sind diese Beweiser nur von speziell geschulten Personen nutzbar, die über Wissen und Erfahrung auf dem Anwendungsgebiet, mit dem Kalkül, der verwendeten Taktiksprache und manchmal sogar der Implementationssprache des Beweisers verfügen. Bekannte Interaktive Theorem-Beweiser sind Isabelle, HOL, Nuprl und Coq, der in diesem Bericht beispielhaft erläutert wird.

2.2.2 Automatische Theorem-Beweiser

Dies sind Programme, die vollautomatisch zu einer gegebenen Formel versuchen zu zeigen, dass diese gültig ist oder nicht. Sie basieren aufgrund der Existenz von effektiven Inferenzregeln und Beweisverfahren meist auf Prädikatenlogik erster Stufe oder einer Teilklasse davon. Die zwei verschiedenen Ansätze, die verwendet werden, um den Beweis zu führen, sind Resolutions- und Tableauverfahren. Beides sind Widerspruchsverfahren, welche die Gültigkeit einer Formel f durch die Unerfüllbarkeit von $\neg f$ zeigen.

Obwohl es mittlerweile relativ weit entwickelte Beweiser dieser Klasse gibt, werden sie kaum in der Praxis eingesetzt. Dies liegt vor allem an zwei Problemen. Selbst bei kleinen Beispielen brauchen sie sehr lange, um einen Beweis zu finden, oder sie finden ihn gar nicht. Somit liegen die meisten real

existierenden Probleme außerhalb ihrer Möglichkeiten. Überdies ist das inhärent niedrige Ausdrucksvermögen der zu Grunde liegenden Logik meist nicht ausreichend. Die natürliche Repräsentation von vielen Anwendungsproblemen und deren Beweise benötigt Logik höherer Ordnung.

2.3 Model-Checking

Model-Checking wurde am Anfang der 80er Jahre entwickelt, um nebenläufige Systeme mit endlichem Zustandsraum zu verifizieren. Dieser Ansatz wird Model-Checking genannt, denn die Idee ist, eine Eigenschaft durch eine Formel in einer vorgegebenen Logik (i.d.R. Temporallogik) darzustellen, sowie das System als Struktur zu sehen, auf der man die Logik als formale Semantik interpretieren kann und überprüft, ob die Struktur ein Modell dieser Formel ist. Somit wird im Gegensatz zum Theorem Beweisen die Aussage, dass das System eine bestimmte Eigenschaft besitzt, dadurch formalisiert, dass die Semantik des Systems ein Model M für die Formel f der Eigenschaft ist.

$$M, s \models f$$

Das übliche Vorgehen beim Model-Checking ist wie folgt: Man erstellt ein spezielles Transitionssystem, Kripke-Struktur genannt, um das Verhalten des Systems zu beschreiben. Die Kripke -Struktur besteht aus einer Menge von Zuständen, einer Menge von Transitionen zwischen den Zuständen und einer Funktion, die jeden Zustand mit einer Menge von atomaren Eigenschaften beschriftet, welche in ihm erfüllt sind. Um nun zu verifizieren, dass das System, dargestellt als Kripke-Struktur M, die Spezifikation, beschrieben durch eine temporallogische Formel f, erfüllt, muss die Menge der Startzustände eine Teilmenge der folgenden, durch den Model-Checking Algorithmus berechneten, Menge sein.

$$\{ s \in S \mid M, s \models f \}$$

Sie enthält alle Zustände, in denen die Formel f wahr ist, und wird durch Analyse des Zustandsraumes gewonnen.

Model-Checking hat, verglichen mit Theorem-Proving, einige Vorzüge. Seine Anwendung erfordert nur geringe Fachkenntnisse und kann somit von einem größeren Personenkreis verwendet werden. Es muss lediglich das System und seine Spezifikation an den Model Checker übergeben werden. Danach überprüft der Algorithmus automatisch die Korrektheit und auf Grund der Endscheidbarkeit der verwendeten Temporallogiken terminiert er und gibt das richtige Resultat aus. Am Resultat der Berechnung sieht man einen weiteren sehr wichtigen Vorteil. Entweder hält der Algorithmus mit einem positiven Ergebnis, oder er gibt ein Gegenbeispiel in Form eines Berechnungspfades, welcher den Fehlerfall charakterisiert. Somit bekommt der Benutzer ein Rückmeldung, die ihn vermuten lässt, an welcher Stelle im System der Fehler vorliegt.

Das größte Problem bei dieser Vorgehensweise ist die Zustandsexplosion, die das exponentielle Wachstum des Zustandsraum beschreibt. Es gibt allerdings viele Ansätze, wie z.B. Ausnutzung von Symmetrien, Verwendung von symbolischen Model-Checking oder Partial Order Reduction, welche im später vorgestellten Model Checker SPIN verwendet wird, um dieses Problem zu mindern. Wie schon eingangs erwähnt, ist es normalerweise nur möglich, Systeme mit einem endlichen Zustandsraum zu verifizieren, was daran liegt, dass ein unendlicher Zustandsraum nicht vollständig durchsucht werden kann. Es gibt allerdings erweiterte Model Checker, die unendliche Zustandsräume mit endlichen Model Eigenschaften verarbeiten können. Da die Verifikation auf der Analyse des gesamten Zustandsraumes basiert und die meisten Tools dieser Methode keinen formalen Beweis liefern, ist es eigentlich nicht möglich, wie beim Theorem-Proving, den Beweis nach seiner Beendigung Korrektur zu lesen. Somit muss man sich auf die Korrektheit der Implementation des Model Checkers verlassen.

Nachdem nun die Hauptcharakteristika, Vorteile und Nachteile der beiden Vorgehensweisen beschrieben wurden, soll nun jeweils ein Beispiel der Theorem-Beweiser und der Model Checker genauer erläutert werden.

2.4 Coq

Das Typsystem Coq ist ein in Objective-Caml geschriebener interaktiver Theorem-Beweiser für Logik höherer Ordnung, der auf dem Predicative Calculus of (Co)Inductive Constructions, einer ausdrucksmächtigen Variante eines getypten λ-Kalküls, basiert. Dieses Beweissystem kann zur Entwicklung von mathematische Theorien und formal zertifizierten Programmen eingesetzt werden und erfolgt in der Sprache Gallina. In ihr wird sowohl die Spezifikation als auch der Beweis formuliert. Im Folgenden werden nun die grundlegenden Konzepte des Kalküls vorgestellt und das Vorgehen bei der Beweisführung erläutert, was an einem Beispiel verdeutlicht wird. Zuletzt wird ein Überblick über die zur Verfügung stehenden Tools und ihre Einsatzmöglichkeiten in der Softwareentwicklung gegeben.

2.4.1 Kalkül

Der Predicative Caluculus of (Co)Inductive Constructions (pCIC) ist die Coq zugrunde liegende formale Sprache. Er ist eine Variante des getypten λ-Kalküls worin jedes Objekt einen Typ besitzt. Jeder Term gehört zu einem Typ, und jeder Typ ist ein Term. Diese Art von Typtheorie kann aufgrund des sogenannten Curry-Howard-Isomorphismus als Logik angesehen werden.

Curry-Howard-Isomorphismus und Intuitionistische Logik

Wie schon erwähnt, beschreibt der Curry-Howard-Isomorphismus die Korrespondenz zwischen Typtheorie und Logik. Der Grundgedanke lässt sich wie folgt beschreiben:

1. Formel = Typ
2. Beweis = Funktion (bzw. Programm)

Die erste Aussage, welche auch unter propositions-as-types bekannt ist, besagt, dass eine logische Formel A interpretiert wird als ein Typ [[A]], so dass A beweisbar ist, wenn es ein Element vom Typ [[A]] gibt. In diesem Sinne wird dann ein Beweis P interpretiert als ein Objekt [[P]], so dass [[P]] ein Element von [[A]] ist. Dies wird durch den zweiten Teil der Korrespondenz ausgedrückt. Der Isomorphismus fordert weiterhin, dass die Normalisierung des Beweises in der Logik der Normalisierung in der Typtheorie entspricht. Um dies zu erreichen, wird im Kalkül eine Intuitionistische Logik verwendet. Da sie eine konstruktive Logik ist, wird gefordert, dass Beweise immer konstruktiv geführt werden. Somit können z.B. Existenzaussagen nicht durch Widerspruch geführt werden, sondern benötigen explizit ein Element (witness) mit den gegebenen Eigenschaften. Das führt bei geeigneten Forderungen an den Grad der Konstruktivität, wie in Coq gegeben, dazu, dass man aus dem Beweis direkt ein Programm extrahieren kann. Coq nutzt die beiden Interpretationen proof-as-types und proof-as-object also so aus, dass das interaktive zielorientierte Beweisen eines Theorems dem Erstellen eines Programmes entspricht. In Intuitionistischer Logik ist die Semantik einer Aussage A nicht, wie in der klassischen Logik, der Wahrheitswert, sondern, wie schon der Curry-Howard-Isomorphismus vermuten lässt, der Raum der Beweise für A. Durch die sogenannte BHK (Brouwer,Heyting und Kolmogorov) Interpretation lässt sich die Intuitionistische Logik wie folgt beschreiben.

1. Ein Beweis für die Konjunktion $A \wedge B$ ist ein Paar mit einem Beweis für A und einem Beweis für B.
2. Ein Beweis für die Disjunktion $A \vee B$ ist ein Paar, bestehend aus einem Beweis für A oder einem Beweis für B zusammen mit Information darüber, was ausgewählt wurde.
3. Ein Beweis für die Implikation $A \Rightarrow B$ ist eine konstruktive Methode, um einen Beweis für B zu liefern, indem man einen Beweis für A findet.
4. Ein Beweis einer universellen Quantifikation $\forall x \in A.B$ ist eine konstruktive Methode, um einen Beweis für $B(a)$ für jedes $x \in A$ zu liefern.
5. Ein Beweis einer existentiellen Quantifikation $\exists x \in A.B$ ist ein Paar, das ein Element aus $a \in A$ und einen Beweis für B enthält.

Durch die proposition-as-types Interpretation wird z.B. $A \wedge B$ durch $A \times B$ und die Implikation $A \Rightarrow B$ als totale Funktion des Typs $A \rightarrow B$ dargestellt.

Ein Beispiel, um den Curry-Howard-Isomorphismus zu erläutern, ist die Funktion

$$curry = \lambda f \lambda x \lambda y.f(x, y)$$

vom Typ $((A \times B) \rightarrow C) \rightarrow A \rightarrow B \rightarrow C)$, die einen Beweis für das Theorem $((A \wedge B) \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C))$ liefert.

Aufgrund dieser Korrespondenz kann Theorem-Proving in Coq als Konstruktion von wohlgetypten Termen und die Überprüfung eines Beweises als Test auf Wohlgetypheit angesehen werden. Der Type-Checking Algorithmus bildet somit den Kern des gesamten Coq-Systems.

Reduktionen

Für die Überprüfung der Wohlgetypheit eines Ausdrucks ist es notwendig, diesen vorher in eine Normalform zu bringen. Um die Darstellung zu erreichen, werden in Coq im Wesentlichen die folgenden, hier nur informell beschriebenen, Reduktionsregeln verwendet. Eine formale Darstellung findet sich in [Coq04a]. Im Weiteren bezeichnet $t\{v/u\}$ die Substitution, welche jedes freie Vorkommen der Variable v im Term t durch den Term u ersetzt.

- *β-Reduktion* transformiert einen β -Redex. Dies ist ein Term der Form $(funv : T \Rightarrow e_1)e_2$, zu einem Term $e_1\{v/e_2\}$.
- *ι-Reduktion* beschäftigt sich mit induktiven Objekten. Sie besagt, dass sich ein Destruktor, der auf ein durch einen Konstruktor erzeugtes Objekt angewendet wird, sich wie erwartet verhält. Diese Reduktion ist dafür zuständig, Berechnungen in rekursiven Programmen sowie numerische Funktionen, wie Addition und Multiplikation, auszuführen.
- *δ-Reduktion* wird benutzt, um Bezeichner durch ihre Definitionen zu ersetzen. Wenn t ein Term und v ein Bezeichner mit dem Wert t' ist, transformiert die δ-Reduktion von v in t den Term t in $t\{v/t'\}$.
- *ζ-Reduktion* ermöglicht es, lokale Bindungen, also eine Formel der Form $letv := e_1 ine_2$, in eine äquivalente Form ohne lokale Bindungen durch $e_2\{v/e_1\}$ zu bringen.

Die Anwendung dieser Regeln im pCIC, insbesondere deren Kombination, besitzt drei wichtige Eigenschaften:

1. *Strong Normalization*: Jede Folge von Reduktionen ist endlich. Dies hat zur Folge, dass jede Berechnung terminiert.
2. *Konfluenz / Church-Rosser*: Wenn es möglich ist, t durch zwei verschiedene Reduktionsfolgen in t_1 und t_2 zu transformieren, dann existiert ein Term t_3, auf den t_1 und t_2 reduziert werden können.
3. *Subject-Reduction*: Wenn t in t' reduziert werden kann und t vom Typ A ist, dann ist auch t' vom Typ A.

Eine wichtige Konsequenz der ersten beiden Eigenschaften ist, dass jeder Term t eine eindeutige Normalform besitzt..

Induktive Datentypen

Ein weiteres wichtiges Konzept im pCIC ist das der Induktiven Datentypen, die eine Erweiterung der von normalen Programmiersprachen zur Verfügung gestellten Datentypen darstellen. Sie können mit rekursiven Typdefinitionen, wie man sie aus funktionalen Programmiersprachen kennt, verglichen werden. Jeder Induktive Datentyp entspricht einer Berechnungsstruktur, die auf Pattern Matching und Rekursion beruht. Durch sie werden rekursive Programmierungen möglich.

Die einfachsten Induktiven Typen sind die aufgezählten (enumerated) Typen, welche verwendet werden, um endliche Mengen zu beschreiben. Ein Beispiel dafür ist der Typ boolean, der wie folgt definiert wird.

```
Inductive bool : Set := true : bool | false : bool
```

Um beliebig große Datenstrukturen zu erstellen, muss Rekursion verwendet werden. Wie dies funktioniert, wird durch die Definition der natürlichen Zahlen veranschaulicht:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat
```

Bei der Erstellung eines Induktiven Datentyps werden durch Coq automatisch Eliminationsregeln erstellt. Sie ermöglichen z.B. die Beweisführung über den Datentyp mit Struktureller Induktion. Zudem erzeugt das System Regeln, die notwendig sind, um rekursiv Funktionen programmieren zu können. Die Struktur einer solchen rekursiven Funktion kann man sich durch die Definition der Addition zweier natürlicher Zahlen verdeutlichen.

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

Obwohl aus dieser beispielhaften Darstellung nicht ersichtlich, unterliegen die Induktiven Definitionen einer Restriktion. Terme eines induktiven Typs werden durch wiederholtes Anwenden des durch die Definition vorgegebenen Konstruktors erzeugt. Sie werden so gebildet, dass es nicht unendlich viele Verzweigungen von Subtermen geben kann. Dies wird durch das mit dem Typ assoziierte Induktionsprinzip ausgedrückt. Um eine solche unendliche Verzweigungsstruktur zu konstruieren, stellt Coq die Co-induktiven Typen zur Verfügung. Ihre Ähnlichkeit zu Induktiven Typen besteht darin, dass auch sie durch wiederholtes Anwenden von Konstruktoren erzeugt werden. Allerdings existieren für sie keine Induktionsprinzipien, und die Verzweigungen können unendlich sein. Ein Beispiel dafür ist ein Stream.

```
CoInductive Stream (A:Set): Set :=
Cons : A -> Stream A -> Stream A.
```

Dieser Typ enthält eine unendliche lange Sequenz von Elementen des Typs A. Streams werden unter anderem dazu verwendet, reaktive Systeme zu modellieren, weil dort meist unendlich lange Ausführungsfolgen vorliegen. Für die rekursive Programmierung gibt es auch hier ein Konstrukt namens CoFixpoint.

2.4.2 Beweisführung in Coq

Wie eingangs erwähnt, ist Coq ein interaktiver Theorembeweiser und somit müssen Inferenzregeln der zugrundeliegenden Logik manuell angewendet werden. Eine solche Inferenzregel kann auf zwei Weisen gelesen und verwendet werden:

1. *Forward Reasoning*: Von den Prämissen zur Konklusion.
   - *Gelesen*: Wenn ich dies und das weiss, dann kann ich jenes schliessen.
   - *Verwendet (Bsp)*: Wenn ich einen Beweis für A habe und einen Beweis für B habe, dann habe ich auch einen Beweis für $A \wedge B$.
2. *Backward Reasoning*: Von der Konklusion zu den Prämisse.
   - *Gelesen*: Um jenes zu beweisen, muss ich dies und das beweisen.
   - *Verwendet (Bsp)*: Um $A \wedge B$ zu beweisen, muss ich A beweisen und B beweisen.

Die Taktiken in Coq verwenden *Backward Reasoning* - sie werden somit auf das zu zeigende Ziel angewendet. Die Bibliothek, die das Beweissystems zur Verfügung stellt, bietet eine umfangreiche Auswahl dieser Taktiken, und, obwohl diese für die meisten Beweise ausreichen, es können noch zusätzliche, durch den Benutzer definierte, Taktiken hinzugefügt werden. Allerdings fällt aufgrund dieser Vielfalt dem Neueinsteiger die Anwendung des Systems recht schwer. Für einen guten Einstieg sei [Coq04b] empfohlen. Selbst kleine Anwendungsbeispiele setzen ein Grundwissen über Coq voraus, dessen Erläuterung über den Rahmen dieses Artikel hinaus gehen würde. Deshalb wird zur Veranschaulichung der Beweisführung die folgende aussagenlogische Tautologie gezeigt:

$$(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$$

1. Nach dem Starten des Beweissystems mittels des Programmes `coqtop` erscheint der Prompt `Coq <`. Zuerst wird der aktuelle Namensraum mit `Section` festgelegt und anschließend werden die verwendeten Variablen deklariert.

   ```
   Section Aussagenlogik.
   Variables A B C : Prop.
   ```

2. Um in den Beweismodus zu gehen, muss das zu beweisende Lemma definiert werden. Aufgrund der forumlas-as-types Interpretation des Curry-Howard-Isomorphismus steht im Lemma statt des Doppelpfeils für die Implikation der Logik der einfache Pfeil des λ -Kalküls. Die Ausgaben über dem Doppelstrich sind die Hypothesen, unter dem Strich steht das aktuelle Ziel.

   ```
   Lemma bsp_tautologie : (A  -> B -> C) -> (A -> B) -> A -> C.

   1 subgoal
   A : Prop
   B : Prop
   C : Prop
   ============================
   (A -> B -> C) -> (A -> B) -> A -> C
   ```

3. Als Nächstes wird die aus dem Kalkül der natürlichen Deduktion bekannte Taktik `intro` (von Introduction) verwendet. Sie beschreibt, wie ein Ziel bewiesen wird, dessen Hauptoperator ein aussagenlogischer Junktor ist. Der zuerst nicht intuitiv erscheinende Effekt, dass durch eine Einführungsregel die Prämisse der Implikation in die Hypothesen genommen wird, liegt am Backward-Reasoning.

```
intro H.

1 subgoal
A : Prop
B : Prop
C : Prop
H : A -> B -> C
============================
(A -> B) -> A -> C
```

4. Durch die Taktik `intros`, kann die Einführungsregel gleich mehrmals angewendet werden.

```
intros.

1 subgoal
A : Prop
B : Prop
C : Prop
H : A -> B -> C
HA : A -> B
H1 : A
============================
C
```

5. Wir können nun unter der Voraussetzung, dass `A` und `B` gelten, mittels der Hypothese `H` unser Ziel `C` beweisen. Dies tun wir mit der Taktik `Apply`.

```
apply H.

2 subgoals
A : Prop
B : Prop
C : Prop
H : A -> B -> C
H0 : A -> B
H1 : A
============================
A
subgoal 2 is:
B
```

6. Da wir im letzten Schritt die Wahrheit von A und B voraussetzen, müssen wir diese nun zeigen. Da `A` direkt in den Hypothesen unter dem Namen `HA` steht, können wir das erste Ziel durch `exact HA` beweisen. Auf das zweite Ziel wenden wir `apply` unter Verwendung der Hypothese `H'` an.

```
exact H. apply H0.

1 subgoal
A : Prop
```

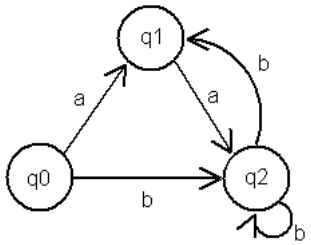


Abbildung 2.1: Zu untersuchendes Transitionssystem

```
B : Prop
C : Prop
H : A -> B -> C
H0 : A -> B
H1 : A
============================
A
```

7. Dieses Ziel könnte mittels `exact HA` bewiesen werden. Doch um deutlich zu machen, dass das aktuelle Ziel sich durch die derzeitigen Hypothesen lösen lässt, verwenden wir `asssumption`.

```
assumption.

Proof completed.
```

8. Der Beweis ist nun abgeschlossen und kann mittels `Save.` der Lemma-Datenbank hinzugefügt werden.

```
Save.

bsp_tautologie is defined
```

Um die Arbeit mit dem Beweissystem zu erleichtern, stellt Coq eine Sammlung von automatischen Taktiken zur Verfügung. Die gerade gezeigte Tautologie hätte sich z.B. auch mit der Taktik `tauto`, welche aussagenlogische Formeln auf Gültigkeit überprüft, automatisch beweisen lassen.

Im Anhang 2.A findet sich ein komplexeres Beispiel für Verifikation mit Coq. Dort wird zum einen gezeigt, dass das in Abb. 2.1 gezeigte Transitionssystem vom Startzustand $q0$ keine Deadlocks enthält, und dass zum anderen unendlich viele bs ausgeführt werden können. Der Leser möge den Beweis mit Hilfe von [Coq04a] nachvollziehen.

2.4.3 Tools

Damit formale Methoden in der Praxis einsetzbar sind, ist es notwendig, dass sie benutzerfreundliche Tools zur Verfügung stellen, die das Arbeiten mit dem Formalismus erleichtern. Coq bietet hierzu textuelle Standardwerkzeuge und die Entwicklungsumgebung CoqIde. Überdies gibt es noch Tools, wie z.B. Krakatoa und Caduceus, die es ermöglichen, Teilklassen der imperativen Programmiersprachen Java bzw. C in Gallina abzubilden.

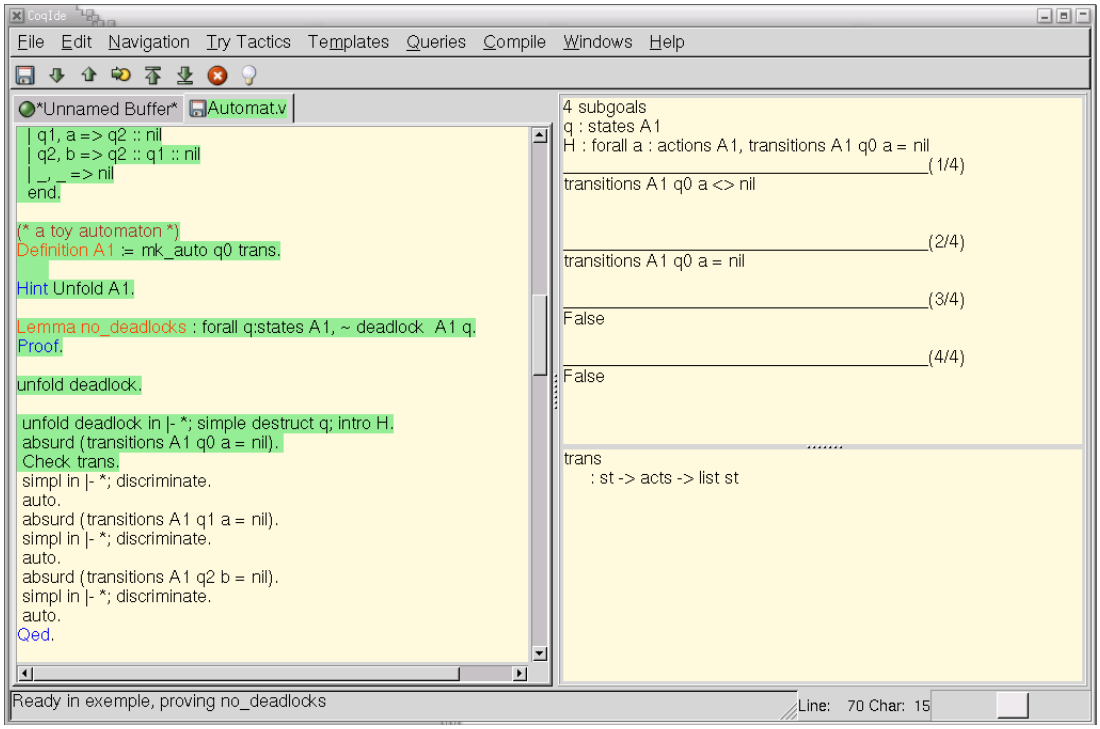


Abbildung 2.2: Die Entwicklungsumgebung CoqIde

Standardwerkzeuge

Normalerweise wird das Beweissystem über das textbasierte Programm *coqtop* benutzt. Der bewiesene Code kann dann mit dem Compiler *coqc* zu einem importierbaren Modul gemacht werden. Zudem gibt auch noch einige Zusatzwerkzeuge, die es z.B. erlauben, den Gallina-Code für Publikationen in ein LaTeX Dokument zu transformieren.

CoqIde

Coq stellt als Entwicklungsumgebung das in Abb. 2.2 gezeigte Werkzeug CoqIDE zur Verfügung. In ihm lassen sich Beweise Schritt für Schritt entwickeln. Auf der linken Seite der Oberfläche kann der Quelltext editiert werden. Die rechte Seite zeigt in der oberen Hälfte die aktuellen Ziele und in der unteren die Ausgaben des Programms. Durch das Menü lässt sich noch ein zusätzliches Anfragefenster öffnen, in dem Befehle eingeben werden können, die den aktuellen Zustand nicht ändern. Überdies sind viele der Zusatzprogramme, wie der LaTeX -Konverter, im Werkzeug integriert.

Krakatoa und Caduceus

Krakatoa und Caduceus sind beides Tools, um eine Teilklasse einer imperativen Programmiersprache in Gallina abzubilden. Auf dieser Grundlage können anschließend Eigenschaften von Programmen nachgewiesen werden. Genau genommen setzen sie den Code nicht direkt in Gallina um, sondern in ein Format, welches vom Programm Why verarbeitet wird. Dieses erzeugt dann die Proof Obligations für Coq.

Krakatoa wird zur Zertifizierung von Java Programmen verwendet. Es wurde in der Praxis bereits eingesetzt, um die Korrekteit von JavaCard Anwendungen zu zeigen. Als Spezifikationssprache be-

nutzt es die Java Modeling Language (JML). Die Verwendung des Programms wird in [MPU04] beschrieben.

Zur Verifikation von C Programmen kann das in [FM04] beschrieben Tool Caduceus benutzt werden. Wie Krakatoa, setzt es auch auf Why auf, benutzt jedoch zur Spezifikation ein spezielles Kommentarformat, das der JML ähnelt.

2.5 SPIN

Spin ist ein von der Industrie eingesetzter Model Checker zur Verifikation von verteilten und nebenläufigen Systemen. Er bietet zur Beschreibung des Systems eine Notation namens PROMELA, die imperativen Programmiersprachen ähnelt, und er verarbeitet als Spezifikation Formeln der Logik LTL (Linear Temporal Logic).

Damit das Problem der Zustandsexplosion vermindert wird, verwendet das Tool Techniken, wie z.B. Partial Order Reduction, welche im Laufe dieses Artikels beschrieben werden.

2.5.1 Das LTL Model-Checking Problem

Da Spin ein Model Checker für die Temporallogik LTL ist, lässt sich das Model-Checking Problem aufgrund der Struktur der Formeln wie folgt spezialisieren:

$$M, s \models Ag$$

In natürlicher Sprache bedeutet es, dass in der Kripkestruktur M vom Startzustand s auf allen Pfaden die eingeschränkte Pfadformel g gilt. Dies ist äquivalent zur Aussage, dass es in der Kripkestruktur M vom Startzustand s keinen Pfad gibt, auf dem nicht die eingeschränkte Pfadformel g gilt:

$$M, s \models \neg E \neg g$$

Dieses Problem ist im Allgemeinen, wie in [SC85] bewiesen wurde, PSPACE-vollständig. Spin setzt daher mehrere Methoden ein, um das Problem für bestimmte Teilklassen abzuschwächen, und ist somit für viele Anwendungsbereiche noch ein einsetzbares Werkzeug. Doch bevor auf deren Funktionsweise eingegangen wird, soll zunächst erläutert werden, wie der allgemeine Algorithmus funktioniert.

2.5.2 Der Algorithmus

In der Spezifikationssprache PROMELA (PROcess MEta LAnguage) werden nebenläufige Systeme durch ein oder mehrere Prozess-Templates beschrieben. Diese stellen das Verhalten des jeweiligen Prozesstyps dar und werden am Anfang der Verifikation in endliche Automaten transformiert. Um das globale Verhalten des nebenläufigen Systems zu erhalten, wird das sogenannte *asynchronous interleaving product* der einzelnen Automaten erstellt. Das Ergebnis dieses Produkts ist der Automat, welcher den gesamten Zustandsraum in Form des Erreichbarkeitsgraphen ausdrückt. Danach wird die LTL Formel, welche die zu untersuchende Eigenschaft beschreibt, in einen Büchi-Automaten transformiert. Man müsste nun zeigen, dass die Sprache des Systems eine Teilmenge der Sprache der Eigenschaft ist. Dafür benötigte man allerdings als obere Schranke für den Speicherbedarf die Größe des kartesischen Produkts des Systems und der Eigenschaft sowie mindestens so viel Platz, wie die Größe der Summe der beiden Automaten. Deswegen werden in Spin keine positiven Eigenschaften spezifiziert, sondern negative - also Fehler - die niemals eintreten dürfen. Möchte man dennoch eine

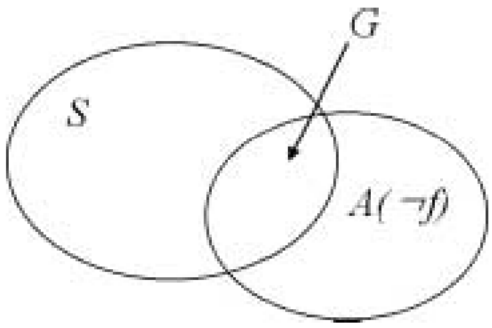


Abbildung 2.3: Durchschnitt G des Systems S und dem Automaten von $\neg f$

positive Aussage beweisen, so muss die Formel vor Übergabe an den Model Checker negiert werden. Das System überprüft somit keine Mengeninklusion, sondern schneidet wie in Abb. 2.3 den Automaten des globalen Systemverhaltens mit dem der Eigenschaft. Das Ergebnis nennt man *synchronous product*. Ist der Schnitt leer, so ist die Eigenschaft im System nie erfüllt. Diese Berechnung des Zustandsraumes hat zwar als obere Schranke bezüglich des Speicherplatzes ebenfalls das Kartesische Produkt, aber im besten Fall ist die Größe gleich Null.

Damit ein Büchi-Automat akzeptiert, muss mindestens einer seiner Endzustände unendlich oft besucht werden. Um dies in einem endlichen System zu erreichen, muss es immer einen Zyklus vorhanden sein. Die Abwesenheit von Fehlern wird somit durch die Abwesenheit von Aktzeptierungszyklen gezeigt. Aufgefunden werden diese Zyklen durch den sogenannten *Nested Depth-First Search Algorithm*, welcher eine Variante des Algorithmus von Tarjan ist, der die starken Zusammenhangskomponenten berechnet. Um einen Akzeptierungszyklus im Erreichbarkeitsgraphen zu finden, muss er einen Zustand q finden, der die folgenden drei Bedingungen erfüllt:

1. q ist ein Endzustand
2. q ist erreichbar vom Startzustand
3. q ist von sich selbst aus erreichbar

Zunächst sucht der Algorithmus mittels Tiefensuche nach einem Endzustand, welcher vom Startzustand erreichbar ist. Hat er einen solchen gefunden, überprüft er in einer weiteren Tiefensuche, ob dieser Zustand von sich selbst aus erreichbar ist. Trifft dies zu, kann der Algorithmus abbrechen und die Konkatenation der Pfade der ersten und zweiten Tiefensuche als Gegenbeispiel zurückgeben. Die Termination nach Auffinden des ersten Gegenbeispiels hat zwar zur Folge, dass nur ausgedrückt wird, dass es *mindestens* ein Gegenbeispiel gibt, doch dafür muss i.d.R. auch nicht der vollständige Zustandsraum konstruiert werden. Diese Eigenschaft wird als *on-the-fly* Verifikation bezeichnet.

2.5.3 Komplexitätsreduktion

Wie schon erwähnt, ist das LTL Model-Checking Problem PSPACE-vollständig. Um trotzdem noch akzeptable Lösungen zu erhalten, ist es möglich, das zu untersuchende Modell durch Abstraktion zu vereinfachen. Da dies aber nur bis zu einem bestimmten Grad funktioniert, müssen zusätzliche Techniken eingesetzt werden, um das Problem der Zustandsexplosion abzuschwächen. Spin verwendet dafür neben der eben beschriebenen Vorgehensweise der *on-the-fly*Verifikation unter anderem noch den Ansatz der *Partial Order Reduction*.

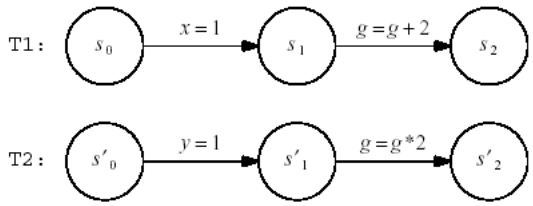


Abbildung 2.4: Automaten T1 und T2

Partial Order Reduction

Diese Methode reduziert die Anzahl der zu untersuchenden Zustände in nebenläufigen Systemen. Sie basiert darauf, dass die Reihenfolgen von unabhängig ausgeführten Aktionen aus Sicht der Spezifikation nicht unterscheidbar sind und deshalb als gleich angenommen werden können. Somit ist es lediglich nötig, einen reduzierten Zustandsraum zu untersuchen, der mindestens einen Repräsentanten für jede Äquivalenzklasse enthält. Diese Technik lässt sich gut durch die in Abb. 2.4 dargestellten Automaten erklären, deren Kanteninschriften Zuweisungen beschreiben. Hier wird jeweils zuerst einer lokalen Variablen ein Wert zugewiesen und anschließend die gemeinsame Variable g geändert.

Die zwei möglichen Reihenfolgen für $x = 1$ und $y = 1$ führen zum gleichen Ergebnis $y = x = 0$. Dies ist hingegen bei $g = g + 2$ und $g = g * 2$ nicht der Fall. Die nebenläufige Ausführung der Automaten führt zu sechs verschiedenen Reihenfolgen der Zuweisungen:

1. $x = 1; g = g + 2; y = 1; g = g * 2;$
2. $x = 1; y = 1; g = g + 2; g = g * 2;$
3. $x = 1; y = 1; g = g * 2; g = g + 2;$
4. $y = 1; g = g * 2; x = 1; g = g + 2;$
5. $y = 1; x = 1; g = g * 2; g = g + 2;$
6. $y = 1; x = 1; g = g + 2; g = g * 2;$

Wie man sieht, unterscheiden sich 1 und 2 nur durch die relative Ordnung der Ausführung von $y = 1$ und $g = g + 2$. Dies sind, genau wie die Sequenzen 4 und 5, sogenannte *unabhängige Operationen*.

Die abhängigen Operationen sind wie folgt:

- $g = g * 2$ und $g = g + 2$
  weil sie auf dasselbe Objekt zugreifen
- $x = 1$ und $g = g + 2$
  weil sie beide in T1 sind
- $y = 1$ und $g = g * 2$
  weil sie beide in T2 sind

und die unabhängigen Operationen:

- $x = 1$ und $y = 1$
- $x = 1$ und $g = g * 2$
- $y = 1$ und $g = g + 2$

Aufgrund dieser Klassifikation können wir die Ausführungsfolgen im System in die zwei Äquivalenzklassen $\{1,2,6\}$ und $\{3,4,5\}$ partitionieren.

Wir können nun die notwendigen Ausführungen auf zwei reduzieren.

$$2:\ x = 1; y = 1; g = g + 2; g = g * 2;$$

$$3:\ x = 1; y = 1; g = g * 2; g = g + 2;$$

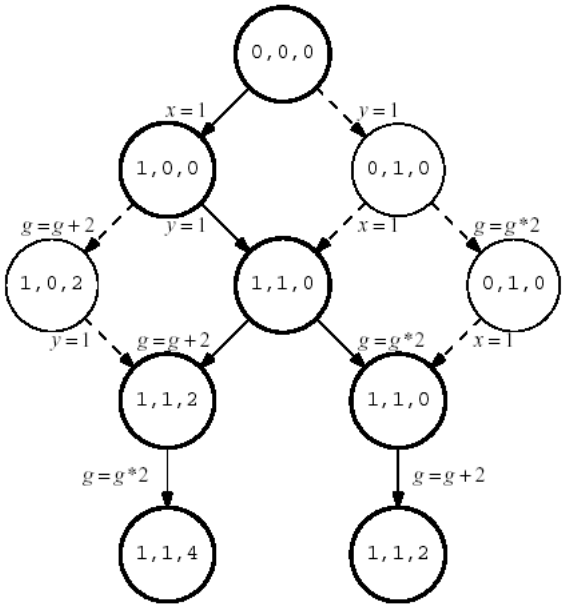


Abbildung 2.5: Vollständige und reduzierte Tiefensuche für T1 $\times$ T2

Alle anderen Ausführungsfolgen können aus diesen beiden durch Permutation der unabhängigen Operationen erlangt werden. In Abb. 2.5 ist der vollständige (durchgezogene Linien, dick gezeichnete Zustände) und reduzierte (gestrichelte Linien, dünn gezeichnete Zustände) Zustandsgraph gezeigt.

2.5.4 Anwendungsbeispiel: Auswahlalgorithmus

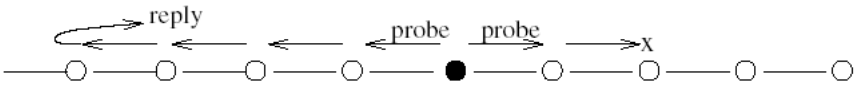


Abbildung 2.6: Senden einer Sonde an 2^l -Nachbarschaft

Aufgrund der Stärken von Spin im Bereich der verteilten Systeme, entschied ich mich für die Modellierung und Verifikation eines Algorithmus, der das Auswahlproblem (Leader Election) für asynchrone, anonyme, nicht uniforme, ungerichtete Ringe, wie in [Val03] beschrieben, mit der Nachrichtenkomplexität $O(nlogn)$ löst. Dort sendet der Knoten p_i in der Phase l eine Nachricht des Typs *Sonde*, wie in Abbildung 2.6 dargestellt, an seine 2^l -Nachbarschaft, wobei die k-Nachbarschaft die Menge der Knoten mit dem maximalen Abstand k ist. Wenn ein Knoten p_j mit $j < i$ diese Nachricht empfängt, dann wird sie entweder weitergesendet, sofern das Ende der 2^l -Nachbarschaft noch nicht erreicht ist. Ansonsten wird eine Reply Nachricht zurückgesendet, die p_i darüber informiert, dass er

der temporäre Leader auf der Seite in Richtung p_j der 2^l -Nachbarschaft ist. Das heißt, dass p_i eine neue Runde beginnen kann, sobald er diese Bestätigung auch aus der anderen Richtung erhält. Wenn p_i in einer Runde seine eigene Sonde empfängt bedeutet dies, dass er die größte Nummer besitzt und nun ausgewählt ist.

Der Algorithmus, dessen PROMELA Modell sich in Anhang 2.B findet, ließ sich in Spin einerseits mittels der programmiersprachenähnliche Notation sehr schnell und andererseits durch die zur Verfügung gestellten Konstrukte auf sehr natürliche Weise modellieren. Ereignisse wie Empfang von getypten Nachrichten auf asynchronen Kanälen ließen sich direkt darstellen. Auch die Verarbeitung und das Senden von Nachrichten bereitete keine Probleme. Nachdem das Modell erstellt war, wurde der Simulator zum Debuggen verwendet. Zudem wurde mit ihm auch überprüft, ob das Modell intuitiv richtig funktioniert. Um einen besseren Einblick in die verteile Anwendung zu bekommen, bietet Spin einem die Möglichkeit, sich ein Nachrichten-Sequenz-Diagramm, wie in Anhang 2.D ersichtlich, anzuzeigen zu lassen, aus dem hervorgeht, wann welche Nachrichten mit welchen Parametern geschickt werden.

Nach ausreichendem Testen des Modells mit dem Simulator erfolgte die Verifikation. Unter Verwendung der folgenden Definitionen wurden die Korrektheitsbedingungen in LTL formuliert:

```
#define elected    (nr_leaders >  0)
#define noLeader   (nr_leaders == 0)
#define oneLeader  (nr_leaders == 1)
```

1. $\Diamond\Box$ oneLeader (notiert: `<>[] noLeader`)
2. $\neg\Diamond$ noLeader (notiert: `![] noLeader`)
3. $\Diamond$ elected (notiert: `<> noLeader`)
4. $\Box$ (noLeader U oneLeader) (notiert: `[] (noLeader U oneLeader)`)

Eine andere Möglichkeit sicherzustellen, dass es genau einen Leader gibt ist es direkt im Modell die folgende Zusicherung zu beschreiben:

```
assert (nr_leaders == 1)
```

2.5.5 Tools

Standardwerkzeuge

Das Paket enthält zum einen das eigentliche Programm `spin`, welches sich über die Kommandozeile mit Parametern steuern lässt, und zum anderen die grafische Oberfläche `XSpin`, die auf `spin` zurückgreift. Sie ist, verglichen mit anderen Model Checkern, sehr benutzerfreundlich und gibt durch kurze Zusammenfassungen im Rahmen der Hilfe einen guten Überblick, wie sich Spin effizient benutzen lässt. Der Simulator bietet viele Einstellungsmöglichkeiten, bleibt aber trotzdem sehr übersichtlich. Viele Konfigurationsoptionen, insbesondere in Bezug auf die Optimierung, bietet auch das Verifikationswerkzeug. Es ist sehr intuitiv zu bedienen und gibt nach abgeschlossener Verifikation eine Statistik bzw. bietet einem die Möglichkeit einer sogenannten *Guided Simulation*, die dem Anwender das Gegenbeispiel unter Verwendung des Simulators darlegt.

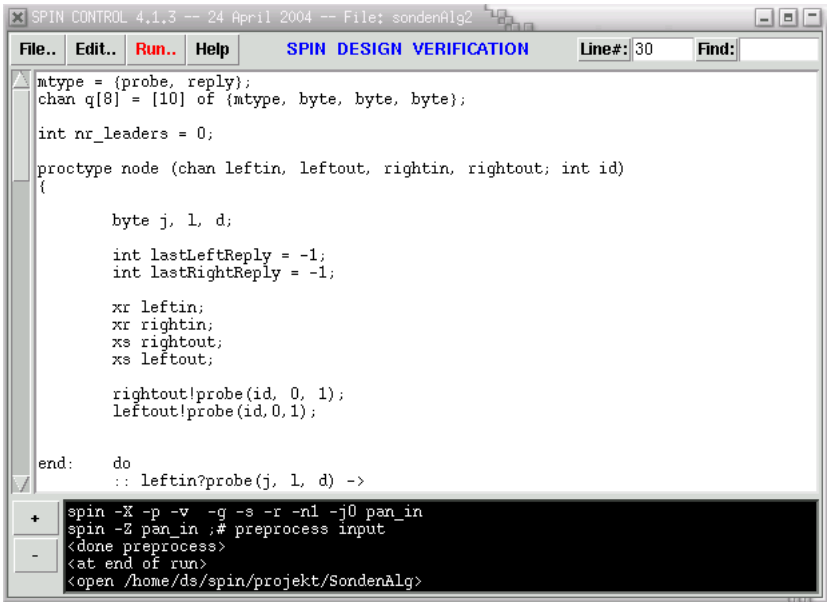


Abbildung 2.7: Die Oberfäche XSpin

Java PathFinder

Das in [HP99]beschriebene, von der NASA entwickelte Programm *Java Pathfinder 1 (JPF1)* übersetzt eine Teilmenge von Java 1.0 in PROMELA. Es soll ein Framework schafften, um Java Programme mittels Model-Checking zu debuggen und zu verifizieren. Eine praxisorientierte Einführung wird in [Hav99] gegeben.

Allerdings unterliegt *JPF1* einigen Restriktionen. So lassen sich manche Elemente von Java nicht darstellen, da PROMELA z.B. für Fließkommazahlen keine Unterstützung bietet. Zudem wird vorausgesetzt, dass alle verwendeten Klassen als Quelltext vorliegen, was bei Gebrauch von Bibliotheken nicht der Fall ist. Zur Lösung diese Problems wurde *JPF2* entwickelt, welches direkt auf Bytecode-Ebene funktioniert und in [HBPV03] vorgestellt wird.

AX

Das Tool *AX*, dessen Abkürzung für *Automaton eXtractor*steht, ist zur Extraktion von Modellen aus der Programmiersprache C gedacht, die sich zur Verifikation mit SPIN eigenen. Der Benutzer kann dabei Einfluss auf den Grad der Abstraktion nehmen. Dieses Werkzeug ist beschrieben in [Hol00].

2.6 Fazit

Es wurden die beiden Hauptrichtungen der Verifikation mit ihren Vor- und Nachteilen erläutert und ein ensprechender Vertreter vorgestellt. Wie sich sehr schnell erkennen ließ, hat die automatisierte Verifikation des Model-Checkings große Vorteile gegenüber dem manuellen Beweisen mittels eines Theorem Provers. Neben der Spezifikationssprache, werden bei der Benutzung nur minimale theoretische Kenntisse durch die Verwendung einer Temporallogik verlangt. Im Falle von Spin kann bei einfachen Eigenschaften selbst diese Forderung durch die programmiersprachlichen Assertions fallen gelassen werden. Die Möglichkeit der Generierung eines Gegenbeispiels stellt einen weiteren erheb-

lichen Vorzug dar. Somit ist Model-Checking, nicht zuletzt auch wegen seiner z.T. sehr ausgereiften Tools, die vorzuziehende Methode für Softwareverifikation.

Im Rahmen der Mathematik ist diese Vorgehensweise jedoch meist nicht anwendbar. Dort muss aufgrund der Beweise über meist unendliche Mengen, wie den reelen Zahlen, Theorem-Proving verwendet werden. Doch in der Softwareentwicklung ist es wegen der zum Teil Monate in Anspruch nehmenden Beweise nur dann sinnvoll, diese Technik zu verwenden, wenn kein Model-Checking anwendbar ist, da die Auswirkung der Zustandsexplosion zu drastisch ist, oder weil unendliche Zustandsräume vorhanden sind. Doch selbst dann sollte vorher ein auf den endlichen Fall begrenztes Modell mit einem Model Checker überprüft werden, denn dies erlaubt die Findung von Fehlern, welche beim Theorem-Proving durch die Unmöglickeit der Umformung in die geforderte Form erst sehr viel später zu Tage treten würden.

Anhang

2.A Transitionssystem in Coq

```
Load LList.
Load LTL.

Hint Unfold satisfies: llists.
Hint Resolve Always_LCons: llists.
Hint Resolve Eventually_here Eventually_further: llists.
Hint Resolve Next_intro: llists.
Hint Resolve Atomic_intro: llists.

Record automaton : Type := mk_auto
  {states : Set;
   actions : Set;
   initial : states;
   transitions : states -> actions -> list states}.

Hint Unfold transitions: automata.

Definition deadlock (A:automaton) (q:states A) :=
  forall a:actions A, @transitions A q a = nil.

Unset Implicit Arguments.
Set Strict Implicit.
CoInductive Trace (A:automaton) : states A -> LList (actions A) -> Prop :=
  | empty_trace : forall q:states A, deadlock A q -> Trace A q LNil
  | lcons_trace :
      forall (q q':states A) (a:actions A) (l:LList (actions A)),
        In q' (transitions A q a) -> Trace A q' l -> Trace A q (LCons a l).
Set Implicit Arguments.
Unset Strict Implicit.


Section Beispielautomat.

(* Zustaende *)
Inductive st : Set :=
  | q0 : st
  | q1 : st
  | q2 : st.
```

```
(* Aktionen *)
Inductive acts : Set :=
  | a : acts
  | b : acts.


(* Transitionen *)
Definition trans (q:st) (x:acts) : list st :=
  match q, x with
  | q0, a => q1 :: nil
  | q0, b => q1 :: nil
  | q1, a => q2 :: nil
  | q2, b => q2 :: q1 :: nil
  | _, _ => nil
  end.

(* Ein kleiner Automat*)
Definition A1 := mk_auto q0 trans.

Hint Unfold A1.

Lemma no_deadlocks : forall q:states A1, ~ deadlock  A1 q.
Proof.
 unfold deadlock in |- *; simple destruct q; intro H.
 absurd (transitions A1 q0 a = nil).
 simpl in |- *; discriminate.
 auto.
 absurd (transitions A1 q1 a = nil).
 simpl in |- *; discriminate.
 auto.
 absurd (transitions A1 q2 b = nil).
 simpl in |- *; discriminate.
 auto.
Qed.

Theorem Infinite_bs :
 forall (q:states A1) (t:LList acts),
   Trace A1 q t -> satisfies t (F_Infinite (Atomic (eq b))).
Proof.
 cofix.
 intros q; case q.
 intros t Ht; case (from_q0 Ht).
 intros x [tx [ea| eb]].
 rewrite ea; split.
 case (from_q2 tx).
 intros x0 [[t2| t1] e]; rewrite e; auto with llists.
 split.
 case (from_q2 tx).
 intros x0 [_ e]; rewrite e; auto with llists.
 apply Infinite_bs with q2; auto.
 rewrite eb; split.
 auto with llists.
 case (from_q2 tx).
 intros x0 [_ e]; rewrite e.
 split.
 auto with llists.
 case e; apply Infinite_bs with q2; auto.
 intros t Ht; case (from_q1 Ht).
```

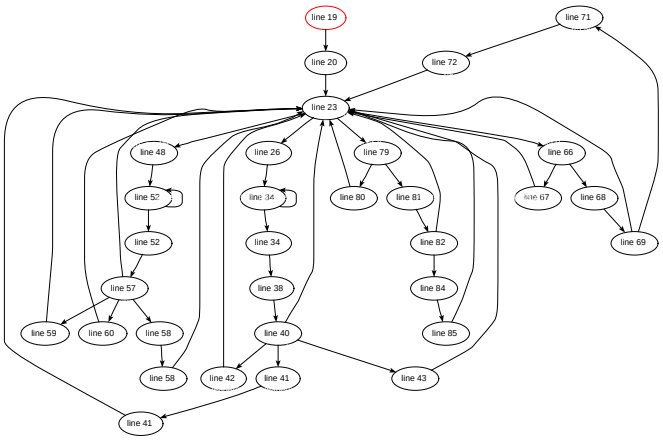
```
 intros x [tx e]; rewrite e.
 split.
 case (from_q2 tx).
 intros x0 [_ e']; rewrite e'; auto with llists.
 apply Infinite_bs with q2; auto.
 intros t Ht; case (from_q2 Ht).
 intros x [[t1| t2] e]; rewrite e.
 split.
 auto with llists.
 apply Infinite_bs with q2; auto.
 split.
 auto with llists.
 apply Infinite_bs with q1; auto.
Qed.

End Beispielautomat.
```

2.B $O(n \cdot \log n)$ Auswahlalgorithmus für asynchrone, nicht-anonyme, ungerichtete Ringe in PROMELA

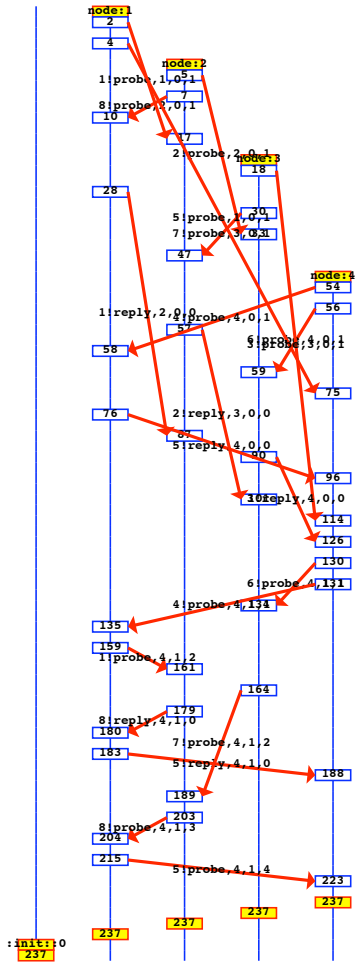
```
mtype = {probe, reply};
chan q[8] = [4] of {mtype, byte, byte, byte};

int nr_leaders = 0;

proctype node (chan leftin, leftout, rightin, rightout; int id)
{

  byte j, l, d;

  int lastLeftReply = -1;
  int lastRightReply = -1;

  xr leftin;
  xr rightin;
  xs rightout;
  xs leftout;

  rightout!probe(id, 0, 1);
  leftout!probe(id,0,1);


end:  do
       :: leftin?probe(j, l, d) ->
               atomic {
                  int pot = 1;
                     int tempL;
                     tempL = l;
                     do
                         ::(tempL != 0) -> pot = pot * 2 ; tempL--
                         ::(tempL == 0) -> break
                     od;
           }
               if
                  ::(j == id) -> nr_leaders++ ; assert(nr_leaders ==1);
```

```
                ::(j > id && d < pot) -> rightout!probe(j,l,d+1)
                ::(j > id && d >= pot) -> leftout!reply(j,l,0)
                :: else
             fi

      :: rightin?probe(j, l, d) ->
             atomic {
                int pot = 1;
                int tempL;
                tempL = l;
                do
                   ::(tempL != 0) -> pot = pot * 2 ; tempL--
                   ::(tempL == 0) -> break
                od;
             }
             if
                ::(j == id) -> nr_leaders++ ; assert(nr_leaders == 1);
                ::(j > id && d < pot) -> leftout!probe(j,l,d+1)
                ::(j > id && d >= pot) -> rightout!reply(j,l,0)
                :: else
             fi

      :: leftin?reply(j, l,0) ->

             if
                ::(j != id) -> rightout!reply(j,l,0);
                ::else -> lastLeftReply = l;
                    if
                       ::(lastLeftReply == lastRightReply) ->
                           leftout!probe(id,l+1,1);
                           rightout!probe(id,l+1,1)
                       :: else
                    fi
             fi

      :: rightin?reply(j,l,0) ->

             if
                ::(j != id) -> leftout!reply(j,l,0);
                :: else -> lastRightReply = l;
                    if
                       ::(lastLeftReply == lastRightReply) ->
                           leftout!probe(id,l+1,1);
                           rightout!probe(id,l+1,1)
                       :: else
                    fi
             fi

   od
}

init {
   atomic {
      run node(q[3], q[4], q[7], q[0], 1);
      run node(q[0], q[7], q[6], q[1], 2);
      run node(q[1], q[6], q[5], q[2], 3);
      run node(q[2], q[5], q[4], q[3], 4);
   }
}
```

2.C Automat des Prozess-Template Node



2.D Nachrichten-Sequenz-Diagramm des Auswahlalgorithmus



Literaturverzeichnis

[Coq04a] Coq. *The Coq Proof Assistant Reference Manual*. LociCal Project, 2004.

[Coq04b] Coq. *The Coq Proof Assistant Tutorial*. LociCal Project, 2004.

[FM04] Jean-Christophe Filliatre and Claude Marche. *The CADUCEUS verification tool for C programs*. 2004.

[Hav99] Klaus Havelund. *Java PathFinder User Guide*. 1999.

[HBPV03] Klaus Havelund, Guillaume Brat, SeungJoon Park, and Willem Visser. *Java PathFinder - Second Generation of a Java Model Checker*. 2003.

[Hol00] Gerald J. Holzmann. Logic verification of ansi-c code with spin. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer-Verlag, 2000.

[HP99] Klaus Havelund and Thomas Pressburger. *Model Checking Java Programs Using Java PathFinder*. 1999.

[MPU04] Claude Marche, Christine Paulin, and Xavier Urbain. *The Krakatoa Tool*. 2004.

[SC85] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. In *Journal of the ACM*, volume 32, pages 733–749. ACM Press, 1985.

[Val03] Rüdiger Valk. Skript zur vorlesung f4 - parallelität und nebenläufigkeit, 2003.

Jens Grabarske
jens@grabarske.de
Fachbereich Informatik
Universität Hamburg

3 Implementation of the spi-calculus

The verification of Computer processes is becoming more and more important. To be able to express concurrent processes the π-calculus by Robin Milner can be used. The spi calculus is a superset of the π-calculus that allows the user to describe the encryption and decryption of messages using either shared keys or public/private key pairs.

The design for an implementation of the spi calculus in Maude is discussed.

Keywords: π calculus, spi calculus, Maude, term rewriting, modelling of processes, security protocols

3.1 Introduction

> If the world was constructed in the same way as computer systems, the first woodpecker coming along would destroy the human civilisation.
>
> *Author unknown*

Although the importance of computer processes is still increasing, even software development companies often fail to realise how important it is to seal their programs against potential attackers.

But even if a company realises the threat coming from the "wild, wild web" – how can a programmer check for leaks in his protocols and methods?

One approach to help software developers in this important but often neglected task is to use automatic provers to verify whether a code fragment obeys certain formal rules which have been specified beforehand. But for this goal a language has to be used in which we can talk about computer processes on an abstract level. The π-calculus, introduced by Robin Milner in 1980 is such a language (Milner1980).

Although the need for such a language was apparently that big that the π-calculus gained a lot of momentum after its introduction, the original specification by Milner did not have a notion for cryptographic communication. This gap was filled by Martín Abadi and Andrew D. Gordon with their so-called *spi-calculus*, a "propped up" version of the π-calculus (Abadi1997).

The task of this student research project was to implement the spi-calculus in Maude, a programming language based on term-rewriting logic, which allows the programmer to shape the syntax of the programs after his own bidding. It was kept in mind that Maude comes with a model checker that can be interfaced with one's own programs – that way the implemented calculus can be used to describe systems that can be checked afterwards.

Section 3.2 describes the syntax and semantics of the π-calculus and the additions made by the spi-calculus. In section 3.3 some of the design decisions are discussed and illustrated. The source code of the system can be found in section 3.A.

3.2 Specifications

3.2.1 The monadic π-calculus

The basic definition of the π-calculus described by Robin Milner is called the *monadic π-calculus* as at each communication step there is only one name that can be sent over the channel.

There are two kinds of entities in the basic π-calculus: *names* and *processes*. A name is just a symbol like a, b or $input$ and stands for a communication channel. A process is a term that describes a certain situation in a system.

Processes can be described in the following manner:

$$P ::= \pi.P \mid P + Q \mid P|Q \mid !P \mid (\nu x)P \mid [x == y]P \mid 0$$
(P and Q are processes, x is a name)

$$\pi ::= x(y) \mid x < y > \mid \tau$$
(π, the name-giver for this calculus, stands for prefixes. In the standard π calculus, there are exactly three kinds of prefixes: $x(y)$, $x < y >$ and τ)

The semantic of these terms is as follows:

Process	**Semantics**
$x < y > .P$	A process that offers to send the name y over the channel x. After it has sent y, it will continue to do P.
$x(y).P$	A process that is waiting for something to be sent via the channel x. When a name is received, it is bound to the name y – meaning that every free instance of y in P is substituted by the message.
$\tau.P$	The τ-process is a "blind" process, something that needs to be processed in one reduction. See the reduction rules below for an example.
$P + Q$	The addition of two processes means that either P or Q can send or receive names.
$P \mid Q$	The par-operator (|) means that P and Q are concurrent. Both P and Q can operate (send or receive names) at a given time.
$!P$	The replication. Semantically it means that there is an arbitrary number of copies of the process P present. One can define the replication recursively with the formula: $!P ::= P\|!P$
$(\nu x)P$	The restriction. The name x will be locally bound in P, meaning, that every free occurence of x in P will be bound.
$[x == y]P$	P can operate if x is bound to y. Otherwise, P is blocked.
0	The "empty" or "idle" process.

The following two formulae define the reduction rules for the monadic π-calculus:

$$\tau.P \Rightarrow P$$
(the hidden operation)

$$((x(y).P) + Q)|((x < z > .R) + S) \Rightarrow P_{\{x/z\}}|R$$
(communication)

The following formulae define some semantical equalities and common syntactic sugar:

$$!P ::= P|!P$$
(definition of the replication)

$$[x == x]P ::= P$$

(semantic of the match-operator)

$$x(y) ::= x(y).0$$
$$x < y > ::= x < y > .0$$

(syntactic sugar)

Some examples

1. $x(y)$
   This process waits for another process to send a message via the communication channel x. As there are no other specified processes, there is no reduction possible.

2. $x(y) \mid x < z >$
   This process consists of two seperate processes, one sending on channel x, the other receiving on x. There is one reduction step possible, with 0 being the outcome of the communication.

3. $x(y).y < z > \mid x < a >$
   This is a more interesting example. After the reduction named "communication" the result is $a < z >$

4. $(x < y > + y < z > | y(z)$
   This term, too, reduces to 0. Only one of the two processes in the sum is selected for the next reduction step. If more than one process in a sum can interact with other (concurrent) processes, then all of these alternatives can be chosen for the further execution of the term. However, only one of these summed processes can communicate with another process.

5. $(\tau.x < y > | x(z)$
   This term needs two reduction steps:
   $(\tau.x < y > | x(z) \Rightarrow x < y > | x(z) \Rightarrow 0$

A more thorough introduction to the monadic and polyadic π-calculus can be found in (Milner1993)

3.2.2 The polyadic π-calculus

The variants of the π-calculus that allow the transmission of more than one name (or instances of other message types) at a time are summarised under the term *polyadic π-calculus*.

The problem, however, is the semantics of this kind of alteration - or, more precisely, the interpretation of what exactly the step from monadicity to polyadicity means. For example, it can be seen as one of the following:

1. **The polyadic π-calculus as syntactic sugar for the monadic π-calculus**
   In this interpretation, a term like $a(bcd)$ is just syntactic sugar for $a(b).a(c).a(d)$.

2. **The polyadicity as a certain demand for the breadth of a message**
   In this interpretation, a term like $a < b\ c\ d > \mid a(s\ t)$ would be an error, because the sending process sends three names, but the receiving process expects only two. Here the polyadicity is interpreted as a negotiated *breadth* of a message. A communicating partner that doesn't adhere to this breadth lets the communication attempt fail.
   Note that the former interpretation would have allowed such a communication ($a < d >$ would have remained).

3. **The polyadicity defines some kind of type system**
   If we take the last interpretation to its extremes, we have some kind of type there – for instance names with the "breadth 2"-type, and so on. This can be used to dictate strict rules for the applications of these types to other channels.

   This kind of behaviour is discussed in (Sangiorgi2001).

For the current implementation, I chose alternative 2.

It is important to point out that even though the latter two alternatives provide some basis for the introduction of types and type checking, strictly spoken they are not exactly needed for the implementation of the spi calculus.

3.2.3 The spi calculus

The spi calculus has the same syntactic elements as the pi calculus, plus the following:

$$[x]_y$$
(the name (message) x is encrypted by the shared key y)

$$case\ x\ of\ [y]_z\ in\ P$$
(if x is an encrypted message using the key z, the content of the decrypted message will be bound to y in P)

$$x-, x+$$
(the private key and the public key of the key pair x)

$$\{[x]\}_y$$
(the message x is encrypted with the public key y)

$$case\ x\ of\ \{[y]\}_z\ in\ P$$
(if x is a message that has been encrypted by a public key whose private key is z, the decrypted message will be bound to y in P)

The operational semantics of the new structures are defined like this:

$$case\ [a]_b\ of\ [c]_b\ in\ P\ ::=\ P_{\{c/a\}}$$
(shared key decryption)

$$case\ \{[a]\}_{b+}\ of\ \{[c]\}_{b-}\ in\ P\ ::=\ P_{\{c/a\}}$$
(private/public key decryption)

Additionally, in the spi calculus a message can be the name of a key, a key pair, a public key, a private key or an encrypted message.

Some examples

1. $x < [y]_z >$
   This process can send a message y on channel x that has been encrypted with the shared key z.

2. $x(m).case\ m\ of\ [y]_k\ in\ a < y >$
   This process can bind a message sent on channel x to the variable m. If m is an encrypted message with key k, the process will continue as a potential sending process on channel a, sending out the formerly encrypted message as clear text.

3. $x < [y]_{z+} > \mid x(m).case\ m\ of\ [a]_{z-}\ in\ b < a >$
   Here two processes can communicate. The first sends a message y that has been encrypted using the public key $z+$. Its communication partner can decrypt the message with the secret key $z-$ and binds the decrypted message to a. The result will be $b < y >$.

3.3 The design

3.3.1 Previous work

In the 15 years of its existence, there have been several implementations of the π-calculus – for instance the so-called Mobility Workbench or the PICT programming language. But there are remarkably few implementations of the spi calculus.

So far there has not been an implementation of the spi calculus in Maude, but the π-calculus was implemented by Prasanna Thati, Koushik Sen and Narciso Martí-Oliet in 2004 (see (Thati2004)). However, I took a slightly different approach on how to treat the reduction rules of the calculus.

Mark-Oliver Stehr described in his thesis (Stehr2002) CINNI, a calculus for name substitutions in arbitrary systems. The code handling the substitutions in my system is an adapted version of the code snippet for CINNI-π he showed there.

3.3.2 Design decisions

In the implementation of the π-calculus and the spi-calculus, there are several difficulties to address, but most importantly the following:

1. **Syntax**
   There are many different notations for the π-calculus, but none is directly translatable to normal ASCII. So one has to choose a notation that can be implemented with just minor alterations.

2. **Substitutions and local variables**
   There are several terms that allow the binding of a local variable or the substitution of a variable with another variable or even other data types. These have to be taken care of.

3. **Order of execution**
   The π calculus implicitly defines a certain order of execution - this has to be translated into Maude.

4. **Replications**
   The semantic of the term $!P$ is that there can be an arbitrary number of instances of P. But how does one decide how many instances actually are needed?

5. **Scalability**
   There are different members of the π calculus family and a good implementation should make it easy to include exactly those features of π calculi, that are needed at a given time.

Syntax

I chose the following conventions in my code:

Element	Syntax
Names	Indexed `Qids` (exception: in binding operators, like the restriction, there only Qids are used), `'x{0}`, `'t{0}`, etc. The index is used by CINNI to distinguish the different instances of a name.
0	`0`
$(\nu x)P$	`(v x) P`
$i(j).0$	`'i{0} ( 'j ) . 0`
$i<j>.0$	`'i{0} < 'j{0} > . 0`
$\tau.0$	`tau . 0`
$P\|Q$, $P+Q$	`P|Q`, `P+Q`
$[x == y]P$	`['x{0} == 'y{0}] P`
$[x]_y$, *case a of* $[b]_c$ *in* P	`['x{0}] 'y{0}`, `case 'a{0} of ['b] 'c{0} in P`
$\{[x]\}_y$, *case a of* $\{[b]\}_c$ *in* P	`{['x{0}} 'y{0}`, `case 'a{0} of {['b]} 'c{0} in P`
$!P$	`!P`

As this table shows, the syntax of my implementation is pretty close to the notation used in the introduction of the calculi.

Substitutions and local variables

Substitutions and local variables are always a little difficult to treat. To accomodate the two problems, Mark-Oliver Stehr invented the so-called "CINNI", *Calculus for Indexed Names and Named Indices*. CINNI itself is more the name of an abstract family of calculi that work with the same principles. These principles applied to a certain language Λ are then called CINNI-Λ.

Stehr gives a couple of Maude implementations for CINNI calculi in his paper as an example of how such a calculus is formed. He also supplied a code snippet for CINNI-π, which I used in my implementation as a starting point.

CINNI-π treats substitutions as first-class objects. They are propagated through the term. Different instances of a variable name are identified using indizes, which are calculated not unlike Berkling indizes.

My implementation handles local variables by first turning them into unique global variables using the indexing mechanism of CINNI-π.

For a more thorough discussion of how CINNI works, see (Stehr2002).

Order of execution

One of the subtleties in the semantics of the π calculus is that the expression $i(y).(a(b)|a < c >)$ cannot reduce to $i(y)$ - i.e.: a communication can only take place at the top level of the expression. To prevent Maude from simply reducing the expression, there are mostly two different approaches:

1. One has to introduce the notion of *locked* processes - this is done in (Thati2004).
2. One has to introduce some kind of top level marker.

I chose the latter alternative. The top level marker is called `tl` and has to be used in every expression that is to be rewritten.

Replications

The naive implementation of the replication operator would be to use its recursive definition – but this would lead to an infinite loop, as Maude would always try to replicate the given process.

But as we do not need to generate a copy of the replicated process unless that process can communicate with another process, I introduced two more rewriting rules in Maude that check whether the process to be replicated could communicate at all – and if so, execute the communication.

Unfortunately, this solution does not prevent infinite loops if the continuation of P itself is a possible communication partner for P or if the continuation can generate a possible communication partner with another replication.

It turned out that these cases are extremely difficult to rule out. As the treatment of endless processes should be allowed in some circumstances, I have not introduced a mechanism to disallow them.

Scalability

The different members of the π calculus family all have one interesting feature: they all have the standard monadic π calculus as a subset in their core. Additionally, many of the features that have been adopted in these variants are orthogonal to each other – for instance polyadicity is orthogonal to the introduction of cryptographic terms as seen in the spi calculus.

This makes the π calculus and the spi calculus ideal candidates to use inheritance. If one writes the Maude modules in such a manner that every new feature of a certain π calculus is orthogonally implemented to the ordinary monadic π calculus, one can define various combinations of advanced features by simply loading the respective modules into Maude.

This was one of the ideas that I used in the design of my system. First I defined the generic syntax of the π calculi by using the notions of *messages* and *bindings* instead of hardwiring the input and output processes to exchange just symbols. On the basis of that syntax definition I implemented CINNI-π, again doing all I could to ensure that it would work also with other π calculi apart from the more basic ones as well. Then I defined the monadic π calculus using the syntactic devices of the other two modules.

3.3.3 Difficulties

Calculus based

The first major hurdle in the implementation of the monadic π calculus was in fact the handling of substitutions and of restrictions. It soon turned out that the problem is not a trivial one. Thankfully, CINNI-π took care of that problem. Actually a lot of π calculus implementations "in the wild" use CINNI-π as a substitution engine, apparently because Stehr found the most practical solution to tackle that problem.

Another problem was the exact semantics of some of the constructs. For instance: What does $x < y > .(y(x)|y < z >)$ mean exactly? Are the two "inner" processes allowed to communicate before the outer process can communicate? Or: when we're asking for a match, like in $[x == y]P$, what exactly does $x == y$ mean? Is it a coreference or true equality? Or is it a congruence? And even if the question is trivial in monadic π-calculus, are there instances of the π-calculus in which it is not trivial?

And, of course, the replication proved to be a complicated issue: is it safe simply to fork a new copy? Do the conditions one introduced under which a new copy can be forked restrict the reduction rules too much? Or too little?

Maude and design based

I came up with the inheritance idea described above not long after I began to work on the project. But while this idea is simple and straight forward enough, it proved to be difficult at times to describe an equation or a reduction rule in such a general way, that it would never need to be touched again in future incarnations of the π calculus.

In fact, forgetting to be general enough in defining the reduction rules was one of the major sources for bugs in implementing the system. I am still convinced, however, that the inheritance idea is "the right way" to implement the π calculus family.

An important aspect of the π calculus is that only top level processes are active. Unfortunately, Maude does not have a notion for the top level itself, so I had to introduce a marker for that.

The more general my definition of the π semantics was, the more difficult it was to introduce syntactic sugar. The more abstract my definitions became, the more confusing syntactic sugar was for Maude - to name just two examples: I could not persuade Maude in the current version of the system to accept `'i{0} < 'j{0} >` as syntactic sugar for `'i{0} < 'j{0} > . 0` or to accept simple Qids as syntactic sugar for a Qid with index 0.

3.4 State of the work and beyond

As far as the project for the seminar "Maschinelles Beweisen" is concernced, the implementation is finished. For the student research project I am going to implement a larger test scenario which I will try to check with the LTL model checker that comes with Maude. Additionally, I will do a more thorough comparison between (Thati2004) and my implementation.

The source code for the system, together with some examples for systems defined with the calculi can be found in the appendix.

Appendix

3.A Source code

Currently there are implementations for

- The monadic π-calculus
- The polyadic π-calculus
- The monadic spi-calculus
- The polyadic spi-calculus

The implementations depend on each other. The files are presented in the order they have to be loaded.

3.A.1 Source files for the monadic π-calculus

channels.maude

```
*** QIDs are ideal candidates for the representation of
*** channel names. Alas, for CINNI-PI, an ordering relation
*** has to be defined on channel names. As the version of
*** QIDs that comes with Maude does not have such a relation
*** per se, ORDERED-QID defines for of them, >, >=, < and
*** <=. It does so by mapping the QID to its string
*** representation and performing a string comparison.
fmod ORDERED-QID is
        protecting QID .
        protecting BOOL .
        protecting STRING .

        *** The operators themselves

        op _<_  : Qid Qid -> Bool .
        op _<=_ : Qid Qid -> Bool .
        op _>_  : Qid Qid -> Bool .
        op _>=_ : Qid Qid -> Bool .

        *** The equations

        *** If X and Y are Qids...
        vars X Y : Qid .

        *** ...then the following equations are tautological

        eq X <  Y = string(X) <  string(Y) .
        eq X <= Y = string(X) <= string(Y) .
        eq X >  Y = string(X) >  string(Y) .
        eq X >= Y = string(X) >= string(Y) .
endfm

*** Apart from the ordering, CINNI-Pi needs to be able
*** to distinguish between different instances of the same
*** channel name. It does so by using indizes. This is
*** implemented by allowing indizes of a QID in this module
fmod INDEXED-VAR is
        protecting ORDERED-QID .
        protecting NAT .

        sort IndexedVar .

        op _{_} : Qid Nat -> IndexedVar [prec 15] .
endfm
```

syntax.maude

```
*** The semantics of the monadic pi calculus cannot be
*** explained without CINNI-PI. CINNI-PI cannot be
*** explained without the syntax of the monadic pi
*** calculus. I solved this dilemma by separating the
*** syntax definition from the definition of the
*** semantics. To make sure that more advanced versions
```

```
*** of the pi calculus can be defined with no real
*** effort, the syntax is defined in the most possible
*** abstract way.
fmod BASIC-PI-SYNTAX is
        protecting INDEXED-VAR .

        *** Channel, Process and Prefix should be
        *** familiar to the reader after the formal
        *** introduction of the pi calculus. This
        *** solution further distinguishes between
        *** Message and Binding to allow for
        *** different types of messages and bindings
        *** in later versions of the pi calculus.
        sorts Channel Process Prefix Message Binding .

        *** All versions of the pi calculus will allow
        *** a basic channel to be both a message and
        *** a channel.
        subsort IndexedVar < Message .
        subsort IndexedVar < Channel .

        *** To tell Maude which expression is to be
        *** reduced next, we need some kind of
        *** execution marker
        op tl_ : Process -> Process [prec 50] .

        *** The "empty" or "idle" process
        op 0 : -> Process [ctor] .

        *** The par-operator (|)
        op _|_ : Process Process -> Process [ctor assoc comm id: 0 prec 45] .

        *** The sum-operator (+)
        op _+_ : Process Process -> Process [ctor assoc comm id: 0 prec 40] .

        *** The prefixes

        *** The prefix-concatenation
        op _._ : Prefix Process -> Process [ctor prec 30] .

        *** in-processes (receiving)
        op _(_) : Channel Binding -> Prefix [ctor prec 25] .

        *** out-processes (sending)
        op _<_> : Channel Message -> Prefix [ctor prec 25] .

        *** case
        op [_==_] : Channel Channel -> Prefix [ctor prec 25] .

        *** tau
        op tau : -> Prefix [ctor] .

        *** replication and restriction
        op v__ : Binding Process -> Process [ctor prec 35] .

        op !_ : Process -> Process [ctor prec 37] .
endfm
```

cinni.maude

```
*** CINNI-PI takes care of substitutions and of telling
*** apart different instances of a certain name (i.e.
*** a "local" and a "global" one)
fmod BASIC-CINNI-PI is
        protecting BASIC-PI-SYNTAX .
        protecting INDEXED-VAR .

        *** CINNI works by treating substitutions as
        *** first-class objects.
        sort Substitution Term .

        subsort Qid        < Binding .
        subsort IndexedVar < Message .

        op [_:=_]    : Binding Message      -> Substitution .
        op [shift_]  : Binding              -> Substitution .
        op [lift__]  : Binding Substitution -> Substitution .

        op __ : Substitution Channel    -> Channel    [prec 15] .
        op __ : Substitution IndexedVar -> IndexedVar [prec 15] .
        op __ : Substitution Prefix     -> Prefix     [prec 15] .
        op __ : Substitution Binding    -> Binding    [prec 15] .
        op __ : Substitution Message    -> Message    [prec 15] .
        op __ : Substitution Process    -> Process    [prec 15] .

        var  S        : Substitution .
        vars n m      : Nat .
        vars X Y Z    : Qid .
        vars CX CY CZ : Channel .
        vars IX IY IZ : IndexedVar .
        vars P Q      : Process .
        vars M N      : Message .
        var  B        : Binding .

        *** Variable renaming

        eq  ([X := N] (X{0}))      = N .
        eq  ([X := N] (X{s(m)}))   = (X{m}) .
        ceq ([X := N] (Y{n}))      = (Y{n}) if X =/= Y .

        *** shifting
        eq  ([shift X] (X{m}))     = (X{s(m)}) .
        ceq ([shift X] (Y{n}))     = (Y{n}) if X =/= Y .

        *** lifting
        eq  ([lift X S] (X{0}))    = (X{0}) .
        eq  ([lift X S] (X{s(m)})) = [shift X] (S (X{m})) .
        ceq ([lift X S] (Y{m}))    = [shift X] (S (Y{m})) if X =/= Y .

        eq  S 0 = 0 .

        ceq S (P | Q) = (S P) | (S Q) if P =/= 0 and Q =/= 0 .
        ceq S (P + Q) = (S P) + (S Q) if P =/= 0 and Q =/= 0 .

        eq S (! P)            = !(S P) .
        eq S (tau . P)        = tau . (S P) .
        eq S (CX < M > . P)   = (S CX) < (S M) > . (S P) .
```

```
        eq S (CX ( B ) . P)   = ((S CX) ( B ) . ([lift B S] P)) .
        eq S ([CX == CY] . P) = ([(S CX) == (S CY)] . (S P)) .
        eq S (v B P)          = v B ([lift B S] P) .
endfm
```

monadic-pi.maude

```
*** Using the notation and the mechanism
*** defined by BASIC-CINNI-PI and the syntax
*** defined by BASIC-PI-SYNTAX, it is now
*** possible to define the semantics and the
*** behaviour of the monadic pi-calculus.
mod MONADIC-PI-CALCULUS is
        protecting BASIC-CINNI-PI .

        vars CX CY CZ    : Channel .
        vars X Y         : Qid .
        vars P Q R S T U : Process .

        var B : Binding .
        var M : Message .

        vars MX MY MZ : Message .
        vars BX BY BZ : Binding .

        vars n m : Nat .

        *** Case elimination.
        eq  ([CX == CX] . P) = P .
        ceq tl (([CX == CY] . P) + Q | R) = tl( Q | R ) if CX =/= CY .

        *** Restriction rewriting

        eq  v X 0 = 0 .
        ceq (v X P) + Q = v X (P + ([shift X] Q)) if Q =/= 0 .
        ceq (v X P) | Q = v X (P | ([shift X] Q)) if Q =/= 0 .

        eq  tl v X P = tl P .
        ceq (v X (v Y P))    = (v Y (v X P)) if Y < X .

        *** The tau-process

        *** (Note that we have to take care to define the reduction
        *** and rewriting steps bearing in mind that there can be
        *** "added" processes or processes, that are in par with
        *** the process in question.)
        rl [tau-elimination] : tl((tau . P) + Q | R) => tl(P | R) .

        *** communication
        rl [communication] : tl(   ((CX < M > . P) + Q)
                                 | ((CX ( B ) . R) + S) | T)

                           => tl(P | ([B := M] R) | T) .

        *** communication with a replicated process
        rl [repl-comm-in]  : tl( ((CX < M > . P) + Q)
                           | !(  ((CX ( B ) . R) + S) | T) | U)
```

```
                        => tl(P | ([B := M] R)    | T
                         | !(((CX ( B ) . R) + S) | T) | U) .

        rl [repl-comm-out] : tl( ((CX ( B ) . P) + Q)
                         | !(  ((CX < M > . R) + S) | T) | U)

                        => tl(([B := M] P) | R    | T
                         | !(((CX < M > . R) + S) | T) | U) .
endm
```

3.A.2 Source files for the polyadic π-calculus

polyadic-pi.maude

```
*** Extending the module MONADIC-PI-CALCULUS to
*** support polyadicity can be a little tricky
*** when one simply defines a list to be of the
*** type message. The reason is that all the
*** rules for the monadic pi calculus would
*** automatically take over - even when the arity
*** of the two communicating processes do not match.
*** That is why POLYADIC-PI-CALCULUS has to be defined
*** in a slightly different way.
mod POLYADIC-PI-CALCULUS is
        protecting MONADIC-PI-CALCULUS .

        sort MessageList .
        sort BindingList .

        *** To facilitate proper type checkings, the lists for
        *** bindings and for messages have to be of two distinct types.

        subsort Message < MessageList .
        subsort Binding < BindingList .

        *** ...but nil can be used for both list types anyway.
        op nil : -> MessageList .
        op nil : -> BindingList .

        op __ : MessageList MessageList -> MessageList [id: nil assoc] .
        op __ : BindingList BindingList -> BindingList [id: nil assoc] .

        *** The polyadic pi-calculus in this interpretation knows
        *** something like a wrong arity of a communicating process.
        *** To signal such an error, the runtime error wrong is
        *** introduced.
        op wrong : -> Process .

        var Bs : BindingList .
        var B  : Binding .
        var Ms : MessageList .
        var M  : Message .
        var CX : Channel .
        vars P Q R T U : Process .
        var S  : Substitution .

        *** applicable? tells us whether a binding list and a message
```

```
        *** list have the same number of elements.
        op applicable? : BindingList MessageList -> Bool .

        eq  applicable?(nil, Ms ) = (Ms == nil) .
        eq  applicable?(Bs , nil) = (Bs == nil) .
        ceq applicable?(B Bs, M Ms) = applicable?(Bs, Ms)
            if B =/= nil and M =/= nil .

        *** syntax extensions (in- and out-processes are allowed to
        *** send / receive lists)
        op _(_) : Channel BindingList -> Prefix [ctor prec 25] .
        op _<_> : Channel MessageList -> Prefix [ctor prec 25] .

        *** CINNI extensions (propagation)
        op __ : Substitution BindingList -> BindingList [prec 15] .
        op __ : Substitution MessageList -> MessageList [prec 15] .

        ceq S (M Ms) = (S M) (S Ms) if Ms =/= nil .

        eq S (CX < Ms > . P) = (S CX) < (S Ms) > . (S P) .
        eq S (CX ( Bs ) . P) = (S CX) ( (S Bs) ) . (S P) .

        *** CINNI extension (the ability to bind a list of values to
        *** a list of names)
        ceq [B Bs := M Ms] P = [Bs := Ms]([B := M] P)
            if Bs =/= nil and Ms =/= nil .

        *** The new rewriting rules - what if the two processes in
        *** question do not have the same arity?
        ceq tl( ((CX < Ms > . P) + Q) | ((CX ( Bs ) . R) + U) | T) = wrong
            if not applicable?(Bs,Ms) .
        ceq tl( ((CX < Ms > . P) + Q) | !(((CX ( Bs ) . R) + U) | T) | U) = wrong
            if not applicable?(Bs,Ms) .
        ceq tl( ((CX ( Bs ) . P) + Q) | !(((CX < Ms > . R) + U) | T) | U) = wrong
            if not applicable?(Bs,Ms) .

        *** If the arities match, the communication is possible
        crl [polyadic-communication] : tl( ((CX < Ms > . P) + Q)
                                         | ((CX ( Bs ) . R) + U) | T)

                                    => tl(P | ([Bs := Ms] R) | T)
            if applicable?(Bs,Ms) .

        crl [pol-repl-comm-in]  : tl( ((CX < Ms > . P) + Q)
                                    | !(((CX ( Bs ) . R) + U) | T) | U)

                                => tl(P | ([Bs := Ms] R) | T
                                    | !(((CX ( Bs ) . R) + U) | T) | U)
            if applicable?(Bs,Ms) .

        crl [pol-repl-comm-out] : tl( ((CX ( Bs ) . P) + Q)
                                    | !(((CX < Ms > . R) + U) | T) | U)

                                => tl(([Bs := Ms] P) | R | T
                                    | !(((CX < Ms > . R) + U) | T) | U)
            if applicable?(Bs,Ms) .
endm
```

3.A.3 Source files for the monadic spi-calculus

monadic-spi.maude

```
*** Using the definition in MONADIC-PI-CALCULUS,
*** the only thing missing for the monadic spi calculus
*** is the treatment for keys and encrypted messages.
*** This is done in this module.
fmod MONADIC-SPI-CALCULUS is
        protecting MONADIC-PI-CALCULUS .

        sorts EncryptedMessage Key PrivateKey PublicKey SharedKey .
        subsort EncryptedMessage Key PublicKey PrivateKey SharedKey < Message .

        subsort IndexedVar < SharedKey .

        subsorts PrivateKey PublicKey SharedKey < Key .

        *** Shared key communication
        op [_]_ : Message IndexedVar -> EncryptedMessage [ctor prec 27] .
        op case_of[_]_in_ : Message Binding SharedKey Process
                            -> Process [ctor prec 27] .

        *** public / private key encryption

        op _+     : IndexedVar        -> PublicKey [ctor prec 17] .
        op _-     : IndexedVar        -> PrivateKey [ctor prec 17] .
        op {[_]}_ : Message Key       -> EncryptedMessage [ctor prec 35] .
        op case_of{[_]}_in_ : Message Binding Key Process
                              -> Process [ctor prec 35] .

        *** Extension of CINNI
        op __ : Substitution Key              -> Key              [prec 15] .
        op __ : Substitution EncryptedMessage -> EncryptedMessage [prec 15] .
        op __ : Substitution PrivateKey       -> PrivateKey       [prec 15] .
        op __ : Substitution PublicKey        -> PublicKey        [prec 15] .
        op __ : Substitution SharedKey        -> SharedKey        [prec 15] .

        vars CX CY   : Channel .
        vars P Q R   : Process .
        vars L M     : Message .
        var  IX      : IndexedVar .
        var  K       : Key .
        var  X       : Qid .
        var  B       : Binding .
        var  S       : Substitution .
        var  PubKey  : PublicKey .
        var  PrivKey : PrivateKey .

        *** CINNI extensions (propagation of substitutions)
        eq S (case L of  [B]K   in P) = case (S L) of  [B]  (S K)
                                        in ([lift B S] P) .
        eq S (case L of {[B]} K in P) = case (S L) of {[B]} (S K)
                                        in ([lift B S] P) .

        eq S  ([M]  K) =  [(S M)]  (S K) .
        eq S ({[M]} K) = {[(S M)]} (S K) .
```

```
        eq S (IX +) = ((S IX) +) .
        eq S (IX -) = ((S IX) -) .

        *** Elimination of decryption-operators
        *** (and the binding of the message)
        eq tl ( ( case [M]K        of  [B]K        in P ) + Q | R )
         = tl ( ([B := M] P) + Q | R ) .
        eq tl ( (case {[M]} (IX +) of {[B]} (IX -) in P ) + Q | R )
         = tl ( ([B := M] P) + Q | R ) .
endfm
```

3.A.4 Source files for the polyadic spi calculus

polyadic-spi.maude

```
*** As the monadic pi calculus, the monadic
*** spi calculus and the polyadic pi calculus
*** were programmed with extensibility in mind,
*** the implementation of the polyadic spi calculus
*** is surprisingly short.
fmod POLYADIC-SPI-CALCULUS is
        protecting POLYADIC-PI-CALCULUS .
        protecting MONADIC-SPI-CALCULUS .
endfm
```

3.A.5 Loader files

The following files can be used to load all necessary modules for a certain calculus.

monadic-pi-loader.maude

```
in channels .
in syntax .
in cinni .
in monadic-pi .
```

monadic-spi-loader.maude

```
in monadic-pi-loader .
in monadic-spi .
```

polyadic-pi-loader.maude

```
in monadic-pi-loader .
in polyadic-pi .
```

polyadic-spi-loader.maude

```
in polyadic-pi-loader .
in monadic-spi .
in polyadic-spi .
```

3.A.6 Examples

monadic-pi-examples.maude

```
in monadic-pi-loader .

fmod MONADIC-PI-EXAMPLES is
        protecting MONADIC-PI-CALCULUS .

        *** Example 1 (simple communication)
        op Alice1 : -> Process .
        op Bob1   : -> Process .

        *** command : rewrite ( tl Alice1 | Bob1 ) .

        *** rewrite in MONADIC-PI-EXAMPLES : tl Alice1 | Bob1 .
        *** rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
        *** result Process: tl 'message{0} < 'output{0} > . 0

        eq Alice1 = 'channel{0} < 'message{0} > . 0 .
        eq Bob1   = 'channel{0} ( 'variable   ) . 'variable{0} < 'output{0} > . 0 .

        *** Example 2 (communication not possible because of different
        ***            instances of the channel name)
        op Alice2 : -> Process .
        op Bob2   : -> Process .

        *** command : rewrite ( tl Alice2 | Bob2 ) .

        *** rewrite in MONADIC-PI-EXAMPLES : tl Alice2 | Bob2 .
        *** rewrites: 28 in 0ms cpu (20ms real) (~ rewrites/second)
        *** result Process: tl 'channel{0}('variable) .
        ***                    'variable{0} < 'output{0}  > . 0 |
        ***                     'channel{1} < 'message{0} > . 0

        eq Alice2 = Alice1 .
        eq Bob2   = v 'channel Bob1 .

        *** Example 3 (selecting a process in a sum)
        op Alice3 : -> Process .
        op Bob3   : -> Process .

        *** command : rewrite ( tl Alice3 | Bob3 ) .

        *** rewrite in MONADIC-PI-EXAMPLES : tl Alice3 | Bob3 .
        *** rewrites: 20 in 10ms cpu (10ms real) (2000 rewrites/second)
        *** result Process: tl 'g{0} < 'z{0} > . 0

        eq Alice3 = 'b{0} < 'g{0} > . 0 .
        eq Bob3   = ('a{0} ( 'x ) . 0) + ('b{0} ( 'y ) . 'y{0} < 'z{0} > . 0) .
endfm
```

monadic-spi-examples.maude

```
in monadic-spi-loader .

fmod MONADIC-SPI-EXAMPLES is
```

```
protecting MONADIC-SPI-CALCULUS .

*** Output and Input

op Output : -> Process .

eq Output  = 'archive{0} < 'secret_document{0} > . 0 .

*** Example 1 (simple example for shared key communication)

*** command : rewrite ( tl Alice1 | Bob1 ) .

*** rewrite in MONADIC-SPI-EXAMPLES : tl Alice1 | Bob1 .
*** rewrites: 59 in 0ms cpu (0ms real) (~ rewrites/second)
*** result Process: tl 'archive{0} < 'spi{0} > . 0

op Alice1   : -> Process .
op Bob1     : -> Process .

eq Alice1 = 'common_channel{0} < [ 'spi{0} ] 'shared_key{0} > . 0 .
eq Bob1   = 'common_channel{0} ( 'message ) .
            (case 'message{0} of [ 'secret_document ] 'shared_key{0}
             in Output) .

*** Example 2 (more complicated example using a key server)

*** command : rewrite ( tl Alice2 | Bob2 | SharedKeyServer ) .

*** rewrite in MONADIC-SPI-EXAMPLES : tl Alice2 | Bob2 | SharedKeyServer .
*** rewrites: 145 in 0ms cpu (0ms real) (~ rewrites/second)
*** result Process: tl ! 'key_request{0} < 'special_key{0} > . 0
***                    | 'archive{0} <'spi{0} > . 0

op Alice2           : -> Process .
op Bob2             : -> Process .
op SharedKeyFetcher : -> Prefix  .
op SharedKeyServer  : -> Process .

eq SharedKeyFetcher = 'key_request{0} ( 'shared_key ) .
eq SharedKeyServer  = !( 'key_request{0} < 'special_key{0} > . 0 ) .

eq Alice2     = SharedKeyFetcher . Alice1 .
eq Bob2       = SharedKeyFetcher . Bob1 .

*** Example 3 (simple private/public key example)

*** command : rewrite ( tl Alice3 | Bob3 ) .

*** rewrite in MONADIC-SPI-EXAMPLES : tl Alice3 | Bob3 .
*** rewrites: 60 in 0ms cpu (0ms real) (~ rewrites/second)
*** result Process: tl 'archive{0} < 'spi{0} > . 0

op Alice3           : -> Process .
op Bob3             : -> Process .

eq Alice3 = 'common_channel{0} < {[ 'spi{0} ]} 'key{0} + > . 0 .
eq Bob3   = 'common_channel{0} ( 'message ) .
            (case 'message{0} of {[ 'secret_document ]} 'key{0} -
                                 in Output) .
```

```
        *** Example 4
        *** (more complicated example with a key server and a keyring)

        *** command :
        *** rewrite ( tl Alice4 | Bob4 | PubKeyServer | SecKeyRing ) .

        *** rewrite in MONADIC-SPI-EXAMPLES : tl Alice4 | Bob4
        ***                                     | PubKeyServer | SecKeyRing .
        *** rewrites: 144 in 0ms cpu (0ms real) (~ rewrites/second)
        *** result Process: tl ! 'keyring_request{0} < 'key{0} > . 0
        ***                  | ! 'pubkey_request{0}  < 'key{0} + > . 0
        ***                  | 'archive{0} < 'spi{0} > . 0

        op Alice4           : -> Process .
        op Bob4             : -> Process .
        op PubKeyServer     : -> Process .
        op SecKeyRing       : -> Process .

        eq Alice4 = 'pubkey_request{0} ( 'pubkey ) .
                    'common_channel{0} < {[ 'spi{0} ]} 'pubkey{0} > . 0 .
        eq PubKeyServer = !('pubkey_request{0} < 'key{0} + > . 0) .

        eq Bob4 = 'keyring_request{0} ( 'keypair ) .
                    'common_channel{0} ( 'message ) .
                  (case 'message{0} of {[ 'secret_document ]} 'keypair{0} -
                                       in Output) .
        eq SecKeyRing = !('keyring_request{0} < 'key{0} > . 0) .
endfm


polyadic-pi-examples.maude

in polyadic-pi-loader .

fmod POLYADIC-PI-EXAMPLES is
        protecting POLYADIC-PI-CALCULUS .

        *** Example 1 (correct application)

        *** command : rewrite ( tl Alice1 | Bob1 ) .

        *** rewrite in POLYADIC-PI-EXAMPLES : tl Alice1 | Bob1 .
        *** rewrites: 104 in 0ms cpu (0ms real) (~ rewrites/second)
        *** result Process: tl 'b{0} < 'c{0} 'd{0} > . 0

        op Alice1 : -> Process .
        op Bob1   : -> Process .

        eq Alice1 = 'a{0} < 'b{0} 'c{0} 'd{0} > . 0 .
        eq Bob1   = 'a{0} ( 'e    'f    'g    ) . 'e{0} < 'f{0} 'g{0} > . 0 .

        *** Example 2 (incorrect application)

        *** command : rewrite ( tl Alice2 | Bob2 ) .

        *** rewrite in POLYADIC-PI-EXAMPLES : tl Alice2 | Bob2 .
        *** rewrites: 20 in 10ms cpu (10ms real) (2000 rewrites/second)
        *** result Process: wrong
```

```
        op Alice2 : -> Process .
        op Bob2   : -> Process .

        eq Alice2 = 'a{0} < 'b{0} 'c{0} 'd{0} 'z{0} > . 0 .
        eq Bob2   = Bob1 .
endfm
```

polyadic-spi-examples.maude

```
in polyadic-spi-loader .

fmod POLYADIC-SPI-EXAMPLES is
        protecting POLYADIC-SPI-CALCULUS .

        *** Example 1 (correct application)

        *** command : rewrite ( tl Alice1 | Bob1 ) .

        *** rewrite in POLYADIC-SPI-EXAMPLES : tl Alice1 | Bob1 .
        *** rewrites: 106 in 0ms cpu (0ms real) (~ rewrites/second)
        *** result Process: tl 'b{0} < 'c{0} + ['d{0}]'b{0} > . 0

        op Alice1 : -> Process .
        op Bob1   : -> Process .

        eq Alice1 = 'a{0} < 'b{0} ('c{0}+) ['d{0}]'b{0} > . 0 .
        eq Bob1   = 'a{0} ( 'e    'f    'g    ) . 'e{0} < 'f{0} 'g{0} > . 0 .

        *** Example 2 (incorrect application)

        *** command : rewrite ( tl Alice2 | Bob2 ) .

        *** rewrite in POLYADIC-PI-EXAMPLES : tl Alice2 | Bob2 .
        *** rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
        *** result Process: wrong

        op Alice2 : -> Process .
        op Bob2   : -> Process .

        eq Alice2 = 'a{0} < 'b{0} 'c{0} ('d{0}-) {['z{0}]} 'key{0} > . 0 .
        eq Bob2   = Bob1 .
endfm
```

Literature

(Abadi1997) Martín Abadi, Andrew D. Gordon, **A Calculus for Cryptographic Protocols - The Spi Calculus**, Proc. 4th ACM Conf. on Computer and Communications Security, ACM, New York, NY, 1997, pp. 36-47.

(McCombs2003) Theodore McCombs, **Maude 2.0 Primer - Version 1.0**

(Milner1980) Milner, R., **A Calculus of Communicating Systems**, Lecture Notes in Computer Science, Volume 92, Springer-Verlag, 1980.

(Milner1993) Milner, R., **The Polyadic π-Calculus: a Tutorial** in F. L. Hamer, W. Brauer and H. Schwichtenberg, editors, **Logic and Algebra of Specification**, Springer-Verlag, 1993.

(Sangiorgi2001) Davide Sangiorgi, David Walker, **The π-calculus - A Theory of Mobile Processes**, Cambridge University Press, 2001

(Stehr2002) Mark-Oliver Stehr, **First-Order Representations of Higher-Order Formalisms: A Calculus of Names and Substitutions** in **Programming, Specification, and Interactive Theorem Proving - Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory**, doctoral thesis at the University of Hamburg

(Thati2004) Prasanna Thati, Koushik Sen, Narciso Martí-Oliet, **An Executable Specification of Asynchronous Pi-Calculus - Semantics and May Testing in Maude 2.0**, in **Electronic Notes in Theoretical Computer Science vol. 71**, Elsevier, 2004

Thilo Mende
0mende@informatik.uni-hamburg.de
Fachbereich Informatik
Universität Hamburg

4

Verifikation des Yahalom-Protokolls

Das Yahalom-Protokoll dient der Verteilung eines gemeinsamen, geheimen Schlüssels mit gegenseitiger Authentifizierung unter Verwendung eines vertauenswürdigen Dritten. Um die Sicherheit eines solchen Protokolls zu garantieren ist eine formale Verifikation nötig, ein möglicher Ansatz dazu ist der Model-Check einer Spezifikation des Protokolls. Mit Hilfe der Rewriting-Engine Maude kann eine solche Spezifikation schon während der Entwicklung getestet werden und diese somit erheblich vereinfachen und beschleunigen, was anhand des Yahalom-Protokolls und unter Einsatz der objektorientierten Erweiterungen zu Maude demonstriert wird. Anschließend wird die entstandene Spezifikation an den für Maude verfügbaren Model-Checker angepasst und auf Sicherheitsprobleme hin untersucht werden.

Keywords: Sicherheitsprotokolle, Model-Checking, Maude, Yahalom

4.1 Einleitung

Beim Einsatz von Software in sicherheitskritischen Bereichen ist ein Testen der Funktionalität alleine oft nicht ausreichend. Sind die Kosten im Fehlerfall ausreichend hoch, sei es durch die Offenlegung von Geschäftsgeheimnissen oder die Gefahr von Menschenleben, ist eine Verifikation wünschenswert bzw. notwendig.

Bei Protokollen zum Austausch von Schlüssel-Material hängt die Sicherheit der damit verschlüsselten Daten neben dem verwendeten Algorithmus, von dem Protokoll zum Schlüsselaustausch ab. Obwohl diese Protokolle in den meisten Fällen nur aus einigen wenigen verschickten Nachrichten bestehen, können unter Umständen sehr einfache Angriffe unentdeckt bleiben. Das bekannteste Beispiel ist der lange Zeit unentdeckte Fehler im Needham-Schroeder Public-Key Protokoll (NSPK) [Low96], der deutlich gemacht hat, dass gravierende Fehler bei der Analyse per Hand und dem Testen übersehen werden können. Am Beispiel des Yahalom-Protokolls, welches zum Verteilen von symmetrischen Schlüsseln unter Einsatz eines vertrauenswürdigen Dritten genutzt wird, sollen die Möglichkeiten der Modellierung mit Maude aufgezeigt werden. Die objektbasierte Programmierung in Maude erleichtert die Spezifikation von nebenläufigen Protokollen, insbesondere ist die Anzahl der teilnehmenden Kommunikationspartner dadurch unerheblich.

Schon während der Entwicklung der Spezifikation kann die Ausführung des Protokolls getestet werden. Dadurch wird der Spezifikationsprozess beschleunigt und es können viele Fehler in dieser Phase eliminiert werden.

Die im Folgenden beschriebene Spezifikation verdeutlicht dies, und wird mit dem in Maude enthalten Model-Checker verifiziert. Dieser kann die Gültigkeit temporal-logischer Formeln über das Systemverhalten während der Ausführung überprüfen.

```
1  mod TEST is
2  protecting CONFIGURATION .
3  protecting QID .
4  sorts Principal Field .
5  subsort Qid  < Oid < Principal .
6  op mtfield : -> Field .
7  op server : -> Principal .
8  op n : Principal -> Field .
9  op from_to_send_ : Principal Principal Field -> Msg .
10 op Agent : -> Cid .
11 op  nc :_ : Field -> Attribute .
12 var A : Principal .
13 var f : Field .
14 crl [rule] :
15         < A : Agent | nc : f >
16 =>
17         < A : Agent | nc : mtfield >
18         from(A)to(server)send(f)
19 if f =/= mtfield
20 .
21 endm
22 set trace on .
23 rew < 'Alice : Agent | nc : n('Alice) > .
```

Abbildung 4.1: Maude Beispielcode

Nach einer kurzen Einführung zu Maude werden grundlegende Eigenschaften von Sicherheitsprotokollen beschrieben. Anschließend werden Teile der Implementierung des Yahalom-Protokolls erläutert, diese wird mit dem in Maude enthaltenen Model-Checker verifiziert.

4.2 Maude

Die Programmiersprache Maude erlaubt es, Termersetzungssysteme zu spezifizieren und diese auszuführen. Termersetzungssysteme sind Mengen von gerichteten Gleichungen, wobei diese oft als „rewriting rules"bezeichnet werden. Diese Regeln beschreiben die nebenläufige Veränderungen von Zuständen, damit eignen sich Termersetzungssysteme sehr gut für die Modellierung objekt-orientierter Systeme.

Maude definiert benannte Module die wiederum andere Module importieren können. In diesen können Sorten und Subsortenbeziehungen definiert werden, die als Typen von Variablen und Operationen dienen. Das Verhalten des Systems wird über Gleichungen und Regeln definiert, wobei diese jeweils von Bedingungen abhängen können. Im Folgenden werden nur die benötigten Eigenschaften kurz erläutert, eine detaillierte Beschreibung ist in [CDE+03] zu finden, dort werden auch die formalen Fundamente von Maude erläutert.

In Abbildung 4.1 ist eine Spezifikation in Maude zu finden, die alle benötigten Funktionen einsetzt. Zunächst wird in Zeile 1 ein Modul definiert. Anschließend werden zwei vorgefertigte Module geladen und die benutzten Sorten definiert. Die Subsortenbeziehung in Zeile 5 erlaubt die Identifikation von Objekten und das Erzeugen von Symbolen vom Typ `Principal` über „Quoted Identifier" wie z.B. „ 'Alice". Anschließend werden zwei Konstanten in Form von nullstelligen Operationen und die einstellige Operation `n` zur Erzeugung von Noncen[1] eingeführt. Die dreistellige Operation `from_to_send_` erzeugt ein Objekt der Sorte `Msg`, diese ist in dem in Zeile 2 importierten Modul definiert und eine Teil der Sorte `Configuration`. Ein anderer Teil dieser Sorte sind Objekte, die ebenfalls

[1] Dies sind zufällige Werte, die keinerlei Information über den Erzeuger enthalten.

in `CONFIGURATION` definiert sind, und aus einem `Objektidentifier (Oid)`, einem `ClassIdentifier (Cid)` und einer `AttributeSet` erzeugt werden. Erzeugt werden die Objekte mittels `< OiD : Cid | AttributeSet >` . Zeile 10 definiert den `Cid` Agent, dessen einziges Attribut die Operation `nc` von `Field` nach `Attribute` ist. In Zeile 12 und 13 werden Variablen eingeführt, die für die bedingte Regel `[rule]` benötigt werden. Dort wird als Vorbedingung ein Objekt vom Typ `Agent` mit einem `Field f` festgelegt, dies wird in einen Agenten mit dem leeren Feld und einer Nachricht vom Agenten an den Server umgeschrieben. Das Ergebnis ist vom Typ `Configuration`, von dieser Sorte ist der Gesamtzustand des objektorientierten Systems. In Zeile 21 endet das Modul `TEST`, anschließend wird noch die ausführliche Ausgabe aktiviert und als Test der Maude-Befehl `rewrite` auf eine Initialmarkierung angewendet.

4.3 Sicherheitsprotokolle

Um Daten zwischen zwei Kommunikationspartnern verschlüsselt zu übertragen, werden Schlüssel benötigt. Bei asymmetrischer Verschlüsselung existiert ein Public-Key, den jeder Kommunikationspartner kennt. Damit verschlüsselte Daten können nur mit dem dazu gehörigen geheimen Private-Key entschlüsselt werden. Bei symmetrischer Verschlüsselung wird zur Ver- und Entschlüsselung der gleiche Schlüssel, auch Secret-Key genannt, eingesetzt. Da asymmetrische Verfahren für die im praktischen Einsatz benötigten Datenmengen zu langsam sind, werden symmetrische Algorithmen eingesetzt. Das Hauptproblem dabei ist es, bei allen Kommunikationspartnern den gleichen, geheimen Schlüssel zu etablieren, ohne diesen unverschlüsselt übertragen zu müssen.

Sicherheitsprotokolle beschreiben den Ablauf der Kommunikation zwischen Kommunikationsteilnehmern, oft Principals genannt, um ein oder mehrere sicherheitsrelevante Ziele zu erreichen. Dazu können neben der Wahrung der Anonymität, Integrität oder Nicht-Abstreitbarkeit[2] vor allem die Authentifizierung und der Austausch von Sitzungsschlüsseln zählen. Als solche werden Schlüssel bezeichnet, die nur ein oder wenige Male zur Kommunikation eingesetzt werden. Ein Angreifer kann so nur wenige mit dem gleichen Schlüssel codierte Daten abfangen, wodurch die Kryptanalyse erheblich schwieriger wird. Des weiteren beeinflusst die Kompromittierung des Sitzungsschlüssels im Idealfall auch nur eine Sitzung und nicht die gesamte Kommunikation. Neben Key-Agreement-Techniken wie dem Diffie-Hellman-Verfahren, wo ein Shared-Secret[3] aus öffentlichen und privaten Informationen abgeleitet werden kann, kommen vor allem so genannte Key-Transport-Verfahren zum Einsatz[4] . Dabei wird der Sitzungsschlüssel mit einem anderen Schlüssel codiert übertragen. Beim Einsatz von asymmetrischen Schlüsseln wird von hybriden Verfahren gesprochen. Dabei ist jedoch eine Public-Key Infrastruktur für die Authentifizierung nötig, außerdem kann dabei unter Umständen ein Protokollteilnehmer den Sitzungsschlüssel vorgeben, wodurch Sicherheitsprobleme entstehen können. Bei einer Vorverteilung von symmetrischen Schlüsseln sind bei n potentiellen Teilnehmern 2^n Schlüssel zu erzeugen und zu verwalten.

Da dies nicht praktikabel ist und auch dabei die Authentifizierung auf andere Weise gesichert werden muss, werden so genannte vertrauenswürdige Dritte eingesetzt. Dies sind spezielle Server, mit denen jeder Teilnehmern vorab einen geheimen Schlüssel austauscht. Sitzungsschlüssel werden dann von einem Kommunikationspartner erzeugt und verschlüsselt an den Server geschickt. Dieser agiert als Key-Translation-Center, entschlüsselt die Nachricht und leitet sie erneut verschlüsselt an den gewünschten Teilnehmer weiter. Wenn ein Teilnehmer dem Server nur den gewünschten Kommunikationspartner mitteilt, agiert dieser als Key-Distribution-Center (KDC). Er erzeugt einen Sitzungsschlüssel und versendet diesen mit den jeweils geheimen Schlüsseln codiert an die Teilnehmer.

[2] „non-Repudation“: d.h es kann bewiesen werden, dass eine Teilnehmer gewisse Aktionen durchgeführt hat.
[3] Ein gemeinsames Geheimnis: Dient der Erzeugung eines gemeinsamen, geheimen Schlüssels dient.
[4] Eine detaillierte Beschreibung der unterschiedlichen Verfahren ist in [MVO96] Kapitel 12 zu finden.

4.3.1 Angriffe auf Sicherheitsprotokolle

Im Folgenden wird eine Auswahl an Angriffsmethoden, die auf Sicherheitsprotokolle möglich sind, vorgestellt. Bei der Analyse von Sicherheitsprotokollen wird stets von perfekter Kryptographie ausgegangen, d.h. ein Angreifer kann bei verschlüsselten Nachrichten nur den Typ der Nachricht erraten oder erkennen, diese jedoch unter keinen Umständen entschlüsseln.

Bei der klassischen Replay-Attacke versendet ein Angreifer ganze Nachrichten oder zusammengesetzte Nachrichtenfragmente aus früheren Protokollläufen und versucht sich so als ein Anderer zu maskieren. Diese Angriffe sind meist wirkungslos gegen Protokolle, die Noncen oder Zeitstempel einsetzen. Eine Verfeinerung stellen Replay-Attacken mit mehreren gleichzeitigen, sich überlappenden Protokollläufen[5] dar, in denen ein Angreifer andere Agenten dazu bringt, bestimmte Nachrichten für ihn zu verschlüsseln. In [Syv94] wird eine Klassifizierung von Angriffen anhand der Anzahl sich überlappender Protokollläufe und des Empfängers der gefälschten Nachricht vorgenommen, wobei im folgenden nur eine interleaving-attack mit 2 parallelen Läufen und eine direkte Replay-Attacke gezeigt werden. Ein bekanntes Beispiel für die Bedeutung von Replay-Attacken ist die lange Zeit unentdeckte, in [Low96] beschriebene Attacke gegen das NSPK-Protokoll.

Neben dem schlimmsten Fall, der Maskierung des Angreifers als ein anderer Agent, kann auch die Verletzung bestimmter Eigenschaften des Sicherheitsprotokolls als erfolgreicher Angriff betrachtet werden. „Key Confirmation“ ist eine solche Eigenschaft und bedeutet, dass ein Agent sicher sein kann, dass sein Kommunikationspartner auch wirklich am Protokolllauf beteiligt war.

4.3.2 Das Yahalom-Protokoll

Das Yahalom-Protokoll beschreibt den Austausch eines geheimen Schlüssels über ein KDC mit gegenseitiger Authentifikation. In der folgenden Beschreibung seien A und B Principals, S der Server , N_a eine von A erzeugte Nonce, K_a der geheime Schlüssel von a, K_{ab} der Session-Key von A und B und $\{\ldots\}_{K_a}$ eine mit dem geheimen Schlüssel von A verschlüsselte Nachricht.

Der Initiator A schickt die Nachricht $\{A, n_a\}$ an B, um einen Sitzungsschlüssel mit B zu etablieren. B ergänzt die Nachricht von A um eine Nonce, verschlüsselt dies mit seinem geheimen Schlüssel und schickt die Nachricht $\{B, \{A, N_a, N_b\}_{Kbs}$ an das KDC. Der Server kann die Nachricht entschlüsseln und schickt die dritte Nachricht $\{B, K_{ab}, N_a, N_b\}_{Kas}\{A, K_{ab}\}_{Kbs}$ an den Initiator. A kann den ersten Teil dieser Nachricht entschlüsseln und erhält damit den Sitzungsschlüssel Kab. Der zweite Teil der dritten Nachricht wird um die mit Kab verschlüsselte Nonce von B ergänzt und an B gesandt, aus der B mit seinem geheimen Schlüssel den Sitzungsschlüssel extrahieren kann. In Abbildung 4.2 ist der Ablauf in graphischer Form zu finden.

Die mitgeschickten Noncen dienen der Unterscheidung verschiedener Protokollläufe und schützen gegen einfache Replay-Attacken. Beim Empfang von Nachricht 3 und 4 müssen die erhaltenen Noncen dazu mit den urspünglich versandten verglichen werden. Das zu verifizierende Protokoll ist die in [BAN89] beschriebene, vereinfachte Variante des Yahalom-Protokolls.

Der Protokollablauf ist in Abbildung 4.3 zu finden.

Die vereinfachte Variante unterscheidet sich vom ursprünglichen Yahalom-Protokoll (siehe Abbildung 4.2) nur durch die Behandlung der Nonce N_b, diese ist in der vereinfachten Version in Nachricht 2 unverschlüsselt und wird in Nachricht 3 und 4 zur Nachricht für B hinzugefügt.

[5] Diese Angriffsart wird auch als „interleaving-attack“ bezeichnet.

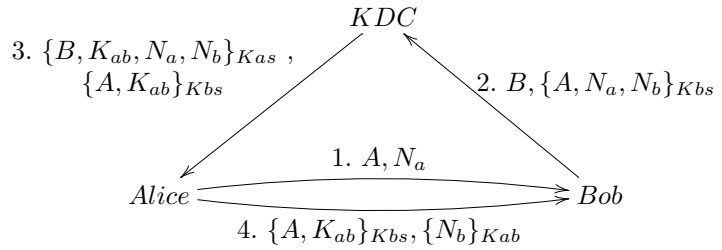


Abbildung 4.2: Das Yahalom Protokoll

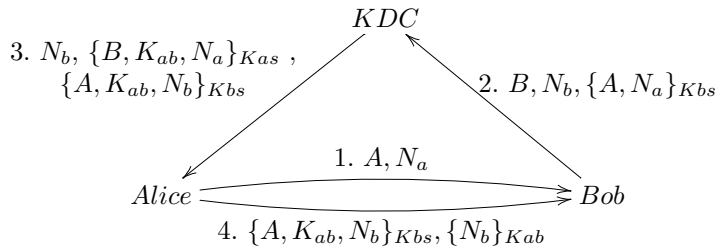


Abbildung 4.3: Die vereinfachte Variante das Yahalom-Protokolls

4.4 Die Spezifikation in Maude

Die hier vorgestellte Spezifikation des vereinfachten Yahalom-Protokolls ist aus der in [NS78] beschriebenen Spezifikation des vereinfachten Needham-Schroeder Public-Key Protokolls entstanden. Zunächst wurde diese an Maude Version 2.1 und die erweiterte objekt-orientierte Syntax angepasst. Anschließend wurde das Grundgerüst um die für das Yahalom-Protokoll benötigten Teile erweitert und der Angreifer an das geänderte Protokoll angepasst. In [NS78] wird eine ebenfalls dort vorgestellte Breitensuche zur Analyse benutzt. Die Eigenschaften der hier entwickelten Spezifikation sollen jedoch mit Hilfe des Model-Checkings verifiziert werden. Die nötigen Anpassungen werden im Abschnitt 4.5.1 beschrieben. Während der Entwicklung traten einige Probleme auf, die zu deutlichen Unterschieden in der Implementierung führten. Diese werden in Abschnitt 4.4.4 detailliert erläutert.

Die drei Dateien `yahalomBAN.maude`, `yahalomBANfixed.maude` und `yahalomBANtypechecking.maude` definieren unterschiedliche Versionen des Protokolls als Modul YAHALOM. Die Dateien `intruder1.maude` und `intruder2.maude` spezifizieren das Verhalten der Angreifer, die grundlegende Implementierung der Angreifer ist, wie auch die der Agenten, in allen Dateien gleich. Die Unterschiede der jeweiligen Spezifikationen werden in Abschnitt 4.5.2 erläutert.

4.4.1 Datentypen und Operationen

Die elementaren Sorten `EstabComm`, `Key`, `Nonce`, `Role`, `Run` und `Principal` sind Subsorten von `Field`. `Principal` dient als ObjectIdentifier und ist eine Obersorte für `Qid`. Solche `QuotedIdentifier` erlauben das Erzeugen von `Principals` über benannte Symbole der Form „'Alice“. Die Operation `from_to_send` verschickt Nachrichten von einem `Principal` an einen Anderen, wobei entweder zwei oder drei `Fields` in der Nachricht enthalten sind. `Field` ist eine Subsorte von Set, `mtset` ist die leere Menge, `add(s1,s2)` vereinigt zwei Mengen, `in(f,s)` prüft, ob ein `Field` f ein Element der Menge s ist. Zur perfekten Ver- und Entschlüsselung dient `ed(Key,Field)`, wobei davon ausgegangen wird, dass `sharedKey(Principal,Principal)` nur vom Server und `secretKey(Principal)` nur vom jeweiligen Principal und vom Server aufgerufen werden.

```
rl [rule3] :
   from(B)to(server)send
      (B)
      (nb)
      (ed(ServerKey(P),cat(A,f)))
=>
   if ( B =/= A )
   then
      from(server)to(A)send
         (nb)
         (ed(ServerKey(A),cat(B,SharedKey(A,B),f)))
         (ed(ServerKey(B),cat(A,SharedKey(A,B),nb)))
   else
      none
   fi
.
```

Abbildung 4.4: Die Regel rule3 aus yahalomBAN.maude

4.4.2 Die Agenten

Die Klasse Agent mit dem Object-Identifier `Principal` und dem Class-Identifier `Agent` hat die folgenden Attribute:

cnt : Nat : Zähler, dient zur Erzeugung der Noncen.

dcom : Set : Menge der gewünschten Kommunikationspartner.

ecom : Set : Menge der etablierten Verbindungen.

rolei : Set : Menge der aktuellen Protokollläufe, in denen der Agent Initiator ist.

roler : Set : Menge der aktuellen Protokollläufe, in denen der Agent Responder ist.

In den Regeln `[rule1]-[rule5]` wird das normale Protokoll-Verhalten der *Agents* und des Servers definiert. Da der Server keinerlei Statusinformationen speichert, sondern nur auf den Secret-Key zugreift, wird dieser nur durch das Umschreiben einer Nachricht in Regel `[rule3]` modelliert. Hier werden als Beispiele für Regeln nur `[rule3]` und Regel `[rule4]` aus yahalomBAN.maude vorgestellt. Die übrigen Regeln sind im Anhang 4.A zu finden. In Abbildung 4.4 wird die Spezifikation des Servers gezeigt: Die von B verschickte Nachricht wird empfangen, der verschlüsselte Teil mit dem geheimen Server-Key entschlüsselt. Dies geschieht implizit durch das Empfangen als (`ed(ServerKey(P)...)`), ein Angreifer kann `serverKey(P)` nicht aufrufen und muss diesen Teil der Nachricht deshalb als verschlüsseltes Feld empfangen. Eine neu erstellte Nachricht enthält die Nonce von B sowie für A und B jeweils einen mit dem geheimen Schlüssel codierten Teil, in dem die Identität des Kommunikationspartners, ein Sessionkey und die jeweilige Nonce übermittelt wird. Diese neue Nachricht wird an A geschickt, sofern A und B nicht identisch sind.

Das Versenden der Nachricht von Schritt 4 des Protokolls ist in Abbildung 4.5 zu finden: A empfängt eine Nachricht vom KDC und versucht die entsprechende Session im Attribut `rolei` zu finden. Gelingt dies, wird die zu dieser Session gespeicherte Nonce mit der verschlüsselt übertragenen Nonce verglichen. Ist dies erfolgreich, wird eine `estabComm` zum Attribut `ecom` hinzugefügt. Die letzte benötigte Nachricht wird an B versandt. Wenn A der am Anfang empfangenen Nachricht keine Session zuordnen kann oder die Noncen nicht übereinstimmen wird die Nachricht verworfen.

```
rl [rule4] :
   < A : Agent | ecom : ES , rolei : add(run(na,B),RI) , AtSet >
   from(server)to(A)send(nb)
              (ed(ServerKey(A),cat(P,sk,f1)))
              (f)
=>
   if ( P == B and na == f1 )
   then
      < A : Agent |  ecom : add(estab(i,na,B,nb,sk),ES) ,
            rolei : RI , AtSet >
      from(A)to(B)send(f)
            (ed(sk,nb))
   else
      < A : Agent | ecom : ES , rolei : RI , AtSet >
   fi
.
```

Abbildung 4.5: Die Regel rule4 aus yahalomBAN.maude

4.4.3 Ein Angreifer

In diesem Abschnitt wird die Funktionalität der Angreifer beschrieben, die konkrete Implementierung ist im Anhang 4.A zu finden. Beide definieren die gleiche Klasse Intruder, die neben den Attributen, die die Agents speichern, noch folgende Attribute beinhaltet:

agents : Set : Alle Agenten, die dem Angreifer bekannt sind.

attack : Nat : das Flag, in dem der Angreifer einen erfolgreichen Angriff signalisiert.

limit : Nat : maximale Anzahl der gefälschten Nachrichten, siehe 4.5.1.

msgs : Set : Menge aller Nachrichtenteile, die ein Angreifer nicht entschlüsseln kann.

ncs : Set : Menge aller Nachrichtenteile, die ein Angreifer entschlüsseln kann.

Beide Angreifer definieren eine Konstante `init`, die die zur späteren Verifikation verwendete Anfangsmarkierung definiert. Diese erzeugt zwei Agenten, Alice und Bob sowie den Angreifer Eve. Das Verhalten der Angreifer unterscheidet sich erheblich, deshalb werden sie auch getrennt beschrieben

Angreifer 1 (Intruder1.maude)

Dieser Angreifer kann sich durch die Regeln `[1intruder]-[5intruder]` wie ein normaler Agent verhalten. Mit den Regeln `[Intercept1]-[Intercept4]` kann der Angreifer die jeweiligen Nachrichten abfangen. Absender und Empfänger werden in `agents` gespeichert, die Nachricht wird zu `msgs` oder `ncs` hinzugefügt. Mit `[replayMsg1 ]-[replayMsg4]` kann der Angreifer Teile abgefangener Nachrichten zu neuen Nachrichten zusammensetzen. Die Regel `[recognizeAttack]` ändert das Flag `attack`, wenn einem Agenten ein erfolgreicher Protokolllauf mit einem anderen Agenten vorgespielt werden kann, ohne dass dieser jemals statt gefunden hat.

Angreifer 2 (Intruder2.maude)

Der zweite Angreifer versucht, durch das geschickte Zusammensetzen von Nachrichten und das Ausnutzen von fehlender Typsicherheit eine Maskierung zu erreichen. Hierdurch steigt die Zahl der möglichen, gefälschten Nachrichten enorm an, weshalb nur die zur Demonstration eines bestimmten

```
crl [recognizeAttack] :
   < E : Intruder |
      attack : 0 ,
      ecom : add(estab(r1,n1,P1,n2,f), ES) ,
      ESet
   >
   < A : Agent |
      ecom : add(estab(r2,n3,P2,n4,f1),ES) ,
      AtSet    >
=>
   < E : Intruder |
      attack : 1 ,
      ecom : add(estab(r1,n1,P1,n2,f), ES) ,
      ESet
   >
   < A : Agent |
      ecom : add(estab(r2,n3,P2,n4,f1),ES) ,
      AtSet
   >
if r1 =/= r2 and P1 == A and P2 =/= E and f == f1
.
```

Abbildung 4.6: Angriffserkennung in intruder2.maude

Angriffs nötigen Regeln implementiert sind. Der Angreifer kann wie oben beschrieben Nachrichten abhören und neu zusammengesetzte Nachrichten verschicken. Die wichtigsten Regeln sind jedoch `[fakeMsg1]` und `[fakeMsg4]`. Erstere setzt zwei abgefangene Noncen zu einer zusammen und benutzt diese für einen neuen Protokolllauf, die andere nutzt eine Nonce zum verschlüsseln einer anderen Nonce.

Die Regel zum Erkennen eines Angriffs ist in Abbildung 4.6 zu finden, wobei überprüft wird, ob der Angreifer sich selbst gegenüber einem Agenten als ein anderer Agent maskieren kann.

4.4.4 Unterschiede zur NSPK-Spezifikation

Die hier vorgestellte Spezifikation basierte auf der in [NS78] vorgestellten Spezifikation des NSPK-Protokolls. Diese wurde zunächst an Maude Version 2.1 angepasst, da sich die Syntax, vor allem die der objekt-orientierten Erweiterungen, geändert hat. In der NSPK-Spezifikation werden nur Nachrichten mit einem Feld verschickt, diese wurden beim Versender als Liste erzeugt, um dann beim Empfänger aufwändig ausgelesen zu werden. Aus diesem Grund wurde die Erweiterung auf Nachrichten mit 2 oder 3 Feldern vorgenommen, was die Verarbeitung der deutlich komplexeren Nachrichten des Yahalom-Protokolls vereinfacht.

Für den Angreifer ist die Sorte „Menge“ nötig, um Informationen über Agenten genau einmal zu speichern. In der NSPK-Spezifikation wurde der Typ `FieldSet`, ein Stack, zum Speichern sämtlicher Objekt-Attribute benutzt, dieser wurde in der hier vorgestellten Spezifikation durch den Typ `Set` ersetzt.

Um die Übersichtlichkeit der Ausgabe zu erhöhen, wurde in der Datei `prelude.maude`, wo unter anderem `Configuration` definiert wird, die Zeile

```
op __ : Configuration Configuration ->
Configuration [ctor config assoc comm id: none] .
```

um die Format-Optionen erweitert:

```
op __ : Configuration Configuration ->
Configuration [ctor config assoc comm id: none format ( d nt d )] .
```

Damit erscheint jeder Teil einer Configuration in einer neuen Zeile.

4.5 Model-Checking

In der Anfangsphase der Entwicklung von Hard- oder Software ist das Testen gut geeignet, grundlegende Fehler zu finden. Zum Ende der Entwicklung treten aufgrund der gestiegenen Komplexität immer subtilere Fehler auf, die z.B. durch Nebenläufigkeit entstehen. Da Testen immer Systemzustände unberücksichtigt lässt, ist nur die Verifikation aller Zustände akzeptabel, vor allem in sicherheitsrelevanten Bereichen. Mit Model-Checking können Aussagen über alle Zustände eines endlichen Systems verifiziert oder falsifiziert werden. Im Gegensatz zu vielen anderen Ansätzen kann die Ausgabe im Fehlerfall direkt zur Identifikation des Fehlers genutzt werden. Die Grundidee ist die Darstellung des Systems als Kripke-Struktur, worüber dann mittels temporal-logischen[6] Formeln Aussagen getroffen werden können, die auf Gültigkeit geprüft werden.[7]

Das Hauptproblem des Model Checking ist die Zustandsexplosion, welches das enorme Wachsen des Zustandsraumes bezeichnet.

4.5.1 Model-Checking mit Maude

Bestandteil von Maude ist ein LTL-Model-Checker. Dieser wird in die zu überprüfende Spezifikation mittels `inc MODEL-CHECKER` eingebunden. Dazu muss das Modul `model-checker.maude` mit dem Befehl `in model-checker` in die Engine geladen werden, dies geschieht automatisch beim Laden des Yahalom-Moduls. Des weiteren müssen die Zustände für den Model-Checker definiert werden, dies geschieht mittels `subsort Configuration < State`. Configuration wird in `prelude.maude` definiert und besteht aus Objekten und Nachrichten, hier also aus ein oder mehreren Agenten und Angreifern sowie den per `from_to_send__` verschickten Nachrichten. Zu verifizierende Eigenschaften werden als Konstanten vom Typ `Prop` definiert, in diesem Fall `op foundAttack : -> Prop .`. Die Gleichung `eq foundAttack = ...` definiert, in welchen Zuständen diese Eigenschaft wahr ist, nämlich genau dann, wenn vorher einmal die Regel `recognizeAttack` ausgeführt werden konnte.

Durch das Fälschen und Wiederholen von Nachrichten entsteht ein System mit unendlichem Zustandsraum, welches nicht mittels Model-Checking verifiziert werden kann. Aus diesem Grund wird vor jeder Ausführung einer solchen Regel überprüft, ob die Variable `limit` größer als Null ist. Ist dies der Fall, wird beim Schalten der Regel `limit` dekrementiert, wodurch die Anzahl der gefälschten oder wiederholten Nachrichten künstlich begrenzt ist.

Die Verifikation einer LTL-Formel erfolgt z.B. mit dem Befehl `red modelCheck(init, [] ⤻ foundAttack) .` Dabei wird überprüft, ob bei der durch `init` definierten Anfangsmarkierung auf allen Pfaden immer nicht `foundAttack` wahr ist, ein Angreifer also keinen Angriff erkennt.

4.5.2 Gefundene Sicherheitsprobleme

Zum Model-Checken von `yahalomBAN.maude` wird zunächst der jeweilige Angreifer geladen. Der Aufruf `red modelCheck(init, []⤻ foundAttack) .` führt beim ersten Intruder zu einem Gegenbeispiel, welches dem in Abbildung 4.7(a) beschriebenen Angriff entspricht.

[6] Im Folgenden wird hierzu „linear temporal logic (LTL)“ verwendet.
[7] Für detaillierte Informationen Ã¼ber Model-Checking siehe [EC99].

$$
\begin{array}{rcl}
\text{i.1. } A \rightarrow I(B) & : & A, N_a \\
\text{ii.1. } I(B) \rightarrow A & : & B, N_a \\
\text{ii.2. } A \rightarrow I(S) & : & A, N'_a, \{B, Na\}_{Kas} \\
\text{iii.2. } I(A) \rightarrow S & : & A, N_a, \{B, Na\}_{Kas} \\
\text{iii.3. } S \rightarrow I(B) & : & N_a, \{A, K_{ab}, N_a\}_{Kbs}, \{B, K_{ab}, N_a\}_{Kas} \\
\text{i.3. } I(S) \rightarrow A & : & N_i, \{B, K_{ab}, N_a\}_{Kas}, \{A, K_{ab}, N_a\}_{Kbs} \\
\text{i.4. } A \rightarrow I(B) & : & \{A, K_{ab}, N_a\}_{Kbs}, N_i K_{ab}
\end{array}
$$

(a)

$$
\begin{array}{rcl}
\text{i.1. } A \rightarrow I(B) & : & A, N_a \\
\text{i.2. } B \rightarrow I(S) & : & B, N_b, \{A, Na\}_{Kbs} \\
\text{ii.1. } I(A) \rightarrow B & : & A, N_a, N_b \\
\text{ii.2. } B \rightarrow I(S) & : & B, N_b, \{A, Na, Nb\}_{Kbs} \\
\text{i.4. } I(A) \rightarrow B & : & \{A, N_a, N_b\} Kbs, \{Nb\}_{Na}
\end{array}
$$

(b)

Abbildung 4.7: Die Angriffe auf des vereinfachte Yahalom-Protokoll.

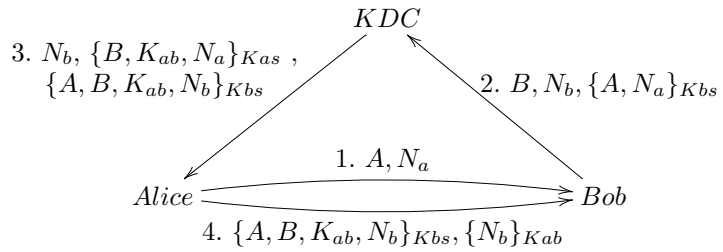


Abbildung 4.8: Gegen den in 4.7(a) beschriebenen Angriff resistente Variante des Yahalom-Protkolls.

Damit kann `intruder1.maude` sich zwar nicht als Angreifer maskieren, aber einem Agenten einen erfolgreichen Schlüsselaustausch mit einem anderen vortäuschen.

Erfolgt der gleiche Aufruf, nachdem `intruder2.maude` geladen wurde, wird der in Abbildung 4.7(b) gezeigte Angriff gefunden. Hier verschickt der Angreifer in Schritt ii.1 zwei zuvor abgehörte Noncen als eine Nonce an B. Dieser verschlüsselt die erhaltene Nachricht und schickt sie verschlüsselt an den Server. Auch diese Nachricht wird abgefangen und kann durch Hinzufügen der mit N_a verschlüsselten Nonce von B als Nachricht 4 der ursprünglichen Kommunikation benutzt werden. Dieser Angriff ist nur möglich, wenn keine Typüberprüfung stattfindet, daher tritt er in yahalomBANTypeChecking.maude nicht auf.

Eine andere Möglichkeit diesen Angriff zu verhindern ist die in Abbildung 4.8 zu findende Variante des Protokolls. Hier wird durch das Hinzufügen der Identität von B in den zweiten Teil von Nachricht 3 eine Asymmetrie zwischen den Nachrichten hergestellt. Diese Änderungen wurden in `yahalomBANfixed.maude` gemacht, ein Model-Check dieser Implementierung ergibt `true`. In diesem Protokoll gilt damit keinem Pfad die Eigenschaft `foundAttack`, die hier vorgestellten Angreifer haben keine Möglichkeit, sich zu maskieren oder einen Schlüsselaustausch vorzutäuschen.

4.6 Fazit

Die Arbeit an der Spezifikation hat gezeigt, dass Maude ein mächtiges Werkzeug zur Modellierung von nebenläufigen, verteilten Systemen ist. Nach einiger Einarbeitung ist schon während der Entwicklung ein Testen möglich, so dass bei der Entwicklung eines Protokolls Fehler frühzeitig auffallen.

Die zweite in Abschnitt 4.5.2 beschriebene Attacke nutzt Typfehler aus. In der ersten Spezifikation[8] konnten diese nicht auftreten, da immer an Felder vom Typ `Key` und `Nonce` gebunden wurde. Erfolgt die eigentliche Implementierung des Protokolls dann in einer nicht typsicheren Sprache, muss die Typsicherheit der Spezifikation entweder nachgebildet werden, oder in der Spezifikation müssen entsprechend generische Typen [9] benutzt werden.

Die hier beschriebenen Angreifer sind stark auf das Yahalom-Protokoll angepasst und ließen sich nicht einfach auf ein anderes Sicherheitsprotokoll adaptieren. Ein sehr allgemeiner Angreifer, der nahezu keine Informationen über das anzugreifende Protokoll benötigt, ist in [Böh02] beschrieben. Dieser erzeugt jedoch keine neue Noncen und hätte die zweite Attacke nicht finden würde, da keine Typfehler ausgenutzt werden.

Zu beachten ist auch, dass die hier beschriebene Verifikation von perfekter Kryptograhie ausgeht. Regeln, die Fehler in den verwendeten Algorithmen oder Timing-Fehler ausnutzen, könnten in Maude implementiert werden. Dann stößt man jedoch schon bei sehr kleinen Protokollen und wenigen gefälschten Nachrichten aufgrund der Zustandsexplosion an die Grenzen des Model-Checkens.

Anhang

4.A Source-Code

yahalomBANTypeChecking.maude

```
1  mod YAHALOM is
2  inc CONFIGURATION .
3  protecting QID .
4  protecting INT .
5  sorts Nonce Key Principal Field Set Run EstabComm Role .
6  subsort Qid < Oid < Principal .
7  subsorts Nonce Principal Run Key EstabComm < Field .
8  subsort Field < Set .
9
10 *** Operationen für Set
11 vars s1 s2 s3 : Set .
12 vars f f1 : Field .
13 op mtset : -> Set .
14 op add : Set Set -> Set [ assoc comm id: mtset ] .
15 op in : Set Set -> Bool .
16
17 eq add(f1,add(f1,s2)) = add(f1,s2) .
18 eq in (f , mtset) = false .
19 eq in(f, add(f1,s3)) = f == f1 or in(f1,s3) .
20
21
22 op n : Principal Nat -> Nonce .
23 op run : Nonce Principal -> Run .
```

[8] siehe `yahalomBANTypeChecking.maude`.
[9] In `yahalomBAN.maude` wird Field dafür eingesetzt.

```
*** Das letze Feld ist vom Typ Key ( vergl. Angriff ü. Typ-Fehler)
op estab : Role Nonce Principal Nonce Key -> EstabComm .
*** einmal mit 2, einmal mit 3 Feldern
op from_to_send__ : Principal Principal Field Field  -> Msg  .
op from_to_send___ : Principal Principal Field Field Field  -> Msg  .
*** Konstante für den Server
op server : -> Principal .
op cat : Field Field -> Field [ assoc ] .

*** Verschlüsselung ( wegen Typ-Sicherheit Key Field -> Field )
op ed : Key Field -> Field .
var k : Key .
eq ed(k,ed(k,f)) = f .
op ServerKey : Principal -> Key .
op SharedKey : Principal Principal -> Key .
ops i r : -> Role .

*** Klassen-ID
op Agent : -> Cid .


*** Objekte-Attribute von Agent
op cnt :_    : Int  -> Attribute .
op dcom :_   : Set  -> Attribute .
op ecom :_   : Set  -> Attribute .
op rolei :_  : Set  -> Attribute .
op roler :_  : Set  -> Attribute .

vars A B P : Principal .
var I : Nat .
vars AS DS ES RI RR : Set .
vars na nc nb : Nonce .
var sk : Key .
var AtSet : AttributeSet .

*** 1. A->B: A, Na
crl [rule1] :
   < A : Agent | cnt : I , dcom : add(B,DS) ,
      rolei : RI , AtSet >
=>
   < A : Agent | cnt : (I + 1) , dcom : DS ,
      rolei : add(run(n(A,I),B),RI) , AtSet >
   from(A)to(B)send(A)(n(A,I))
if B =/= mtset .

*** 1. A->B: A, Na
*** 2. B->S: B,Nb,{A,Na}Kbs
rl [rule2] :
   < B : Agent | cnt : I , roler : RR , AtSet >
   *** hier entsteht TypSicherheit!
   from(P)to(B)send(A)(na)
=>
   < B : Agent | cnt : (I + 1) , roler : add(run(n(B,I),A),RR) , AtSet >
   from(B)to(server)send(B)(n(B,I))(ed(ServerKey(B),cat(A,na)))
.

*** 2. B->S: B,Nb,{A,Na}Kbs
*** 3. S->A: Nb,{B,Kab.Na}Kas,{A,Kab.Nb}Kbs
rl [rule3] :
```

```
   from(B)to(server)send(B)(nb)(ed(ServerKey(P),cat(A,na)))
=>
   if ( B =/= A )
   then
      from(server)to(A)send(nb)
         (ed(ServerKey(A),cat(B,SharedKey(A,B),na)))
         (ed(ServerKey(B),cat(A,SharedKey(A,B),nb)))
   else
      none
   fi
.

*** 3. S->A: Nb,{B,Kab,Na}Kas,{A,Kab,Nb}Kbs
*** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
rl [rule4] :
   < A : Agent | ecom : ES ,
      rolei : add(run(na,B),RI) , AtSet >
   from(server)to(A)send(nb)(ed(ServerKey(A),cat(P,sk,nc)))(f)
=>
   if ( P == B and na == nc )
   then
      < A : Agent | ecom : add(estab(i,na,B,nb,sk),ES) ,
         rolei : RI , AtSet >
         from(A)to(B)send(f)(ed(sk,nb))
   else
      < A : Agent | ecom : ES ,
         rolei : RI , AtSet >
   fi
.

*** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
rl [rule5] :
   < B : Agent | ecom : ES ,
      roler : add(run(nc,P),RR) , AtSet >
   *** SK erzeugt Typsicherheit
   from(A)to(B)send(ed(ServerKey(B),cat(A,sk,nb)))(ed(sk,nb))
=>
   if( nc == nb and P == A )
   then
      < B : Agent | ecom : add(estab(r,nc,A,nc,sk),ES) ,
         roler : RR , AtSet >
   else
      < B : Agent | ecom : ES ,
         roler : RR , AtSet >
   fi
.
endm
```

yahalomBAN.maude

```
mod YAHALOM is
inc CONFIGURATION .
protecting QID .
protecting INT .
sorts Nonce Key Principal Field Set Run EstabComm Role .
subsort Qid < Oid < Principal .
subsorts Nonce Principal Run Key EstabComm < Field .
subsort Field < Set .
```

```
9
10 *** Operationen für Set
11 vars s1 s2 s3 : Set .
12 vars f f1 : Field .
13 op mtset : -> Set .
14 op add : Set Set -> Set [ assoc comm id: mtset ] .
15 op in : Set Set -> Bool .
16
17 eq add(f1,add(f1,s2)) = add(f1,s2) .
18 eq in (f , mtset) = false .
19 eq in(f, add(f1,s3)) = f == f1 or in(f1,s3) .
20
21
22 op n : Principal Nat -> Nonce .
23 op run : Nonce Principal -> Run .
24 *** Das letze Feld ist vom Typ Field (Angriff über Typ-Fehler)
25 op estab : Role Nonce Principal Nonce Field -> EstabComm .
26 *** einmal mit 2, einmal mit 3 Feldern
27 op from_to_send__ : Principal Principal Field Field  -> Msg  .
28 op from_to_send___ : Principal Principal Field Field Field  -> Msg  .
29 *** Konstante für den Server
30 op server : -> Principal .
31 op cat : Field Field -> Field [ assoc ] .
32
33 *** Verschlüsselung ( wegen Typ-Fehler Field Field -> Field )
34 op ed : Field Field -> Field .
35 var k : Key .
36 eq ed(f1,ed(f1,f)) = f .
37 op ServerKey : Principal -> Key .
38 op SharedKey : Principal Principal -> Key .
39 ops i r : -> Role .
40
41 *** Klassen-Id
42 op Agent : -> Cid .
43
44
45 *** Objekte-Attribute von Agent
46 op cnt :_    : Int  -> Attribute .
47 op dcom :_   : Set  -> Attribute .
48 op ecom :_   : Set  -> Attribute .
49 op rolei :_  : Set  -> Attribute .
50 op roler :_  : Set  -> Attribute .
51
52 vars A B P : Principal .
53 var I : Nat .
54 vars AS DS ES RI RR : Set .
55 vars na nc nb : Nonce .
56 var sk : Key .
57 var AtSet : AttributeSet .
58
59 *** 1. A->B: A, Na
60 crl [rule1] :
61    < A : Agent | cnt : I , dcom : add(B,DS) ,
62       rolei : RI , AtSet >
63 =>
64    < A : Agent | cnt : (I + 1) , dcom : DS ,
65       rolei : add(run(n(A,I),B),RI) , AtSet >
66    from(A)to(B)send(A)(n(A,I))
67 if B =/= mtset .
```

```
68
69  *** 1. A->B: A, Na
70  *** 2. B->S: B,Nb,{A,Na}Kbs
71  rl [rule2] :
72     < B : Agent | cnt : I , roler : RR , AtSet >
73     *** eigentlich na, aber dann typsicher
74     from(P)to(B)send(A)(f)
75  =>
76     < B : Agent | cnt : (I + 1) , roler : add(run(n(B,I),A),RR) , AtSet >
77     from(B)to(server)send(B)(n(B,I))(ed(ServerKey(B),cat(A,f)))
78  .
79
80  *** 2. B->S: B,Nb,{A,Na}Kbs
81  *** 3. S->A: Nb,{B,Kab.Na}Kas,{A,Kab.Nb}Kbs
82  rl [rule3] :
83     from(B)to(server)send(B)(nb)(ed(ServerKey(P),cat(A,f)))
84  =>
85     if ( B =/= A )
86     then
87        from(server)to(A)send(nb)
88           (ed(ServerKey(A),cat(B,SharedKey(A,B),f)))
89           (ed(ServerKey(B),cat(A,SharedKey(A,B),nb)))
90     else
91        none
92     fi
93  .
94
95  *** 3. S->A: Nb,{B,Kab,Na}Kas,{A,Kab,Nb}Kbs
96  *** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
97  rl [rule4] :
98     < A : Agent | ecom : ES ,
99        rolei : add(run(na,B),RI) , AtSet >
100    from(server)to(A)send(nb)(ed(ServerKey(A),cat(P,sk,f1)))(f)
101 =>
102    if ( P == B and na == f1 )
103    then
104       < A : Agent | ecom : add(estab(i,na,B,nb,sk),ES) ,
105          rolei : RI , AtSet >
106          from(A)to(B)send(f)(ed(sk,nb))
107    else
108       < A : Agent | ecom : ES ,
109          rolei : RI , AtSet >
110    fi
111 .
112
113 *** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
114 rl [rule5] :
115    < B : Agent | ecom : ES ,
116       roler : add(run(nc,P),RR) , AtSet >
117    *** f ist der Shared-key
118    from(A)to(B)send(ed(ServerKey(B),cat(A,f,nb)))(ed(f,nb))
119 =>
120    if( nc == nb and P == A )
121    then
122       < B : Agent | ecom : add(estab(r,nc,A,nc,f),ES) ,
123          roler : RR , AtSet >
124    else
125       < B : Agent | ecom : ES ,
126          roler : RR , AtSet >
```

```
127     fi
128 .
129 endm
```

yahalomBANfixed.maude

```
1  mod YAHALOM is
2  inc CONFIGURATION .
3  protecting QID .
4  protecting INT .
5  sorts Nonce Key Principal Field Set Run EstabComm Role .
6  subsort Qid < Oid < Principal .
7  subsorts Nonce Principal Run Key EstabComm < Field .
8  subsort Field < Set .
9
10 *** Operationen für Set
11 vars s1 s2 s3 : Set .
12 vars f f1 : Field .
13 op mtset : -> Set .
14 op add : Set Set -> Set [ assoc comm id: mtset ] .
15 op in : Set Set -> Bool .
16
17 eq add(f1,add(f1,s2)) = add(f1,s2) .
18 eq in (f , mtset) = false .
19 eq in(f, add(f1,s3)) = f == f1 or in(f1,s3) .
20
21
22 op n : Principal Nat -> Nonce .
23 op run : Nonce Principal -> Run .
24 *** Das letze Feld ist vom Typ Field  vergl. Angriff ü. Typ-Fehler)
25 op estab : Role Nonce Principal Nonce Field -> EstabComm .
26 *** einmal mit 2, einmal mit 3 Feldern
27 op from_to_send__ : Principal Principal Field Field  -> Msg  .
28 op from_to_send___ : Principal Principal Field Field Field  -> Msg  .
29 *** Konstante für den Server
30 op server : -> Principal .
31 op cat : Field Field -> Field [ assoc ] .
32
33 *** Verschlüsselung ( wegen Typ-Fehler Field Field -> Field )
34 op ed : Field Field -> Field .
35 var k : Key .
36 eq ed(f1,ed(f1,f)) = f .
37 op ServerKey : Principal -> Key .
38 op SharedKey : Principal Principal -> Key .
39 ops i r : -> Role .
40
41 *** Klassen-Id
42 op Agent : -> Cid .
43
44
45 *** Objekte-Attribute von Agent
46 op cnt :_    : Int  -> Attribute .
47 op dcom :_   : Set  -> Attribute .
48 op ecom :_   : Set  -> Attribute .
49 op rolei :_  : Set  -> Attribute .
50 op roler :_  : Set  -> Attribute .
51
52 vars A B P : Principal .
```

```
53  var I : Nat .
54  vars AS DS ES RI RR : Set .
55  vars na nc nb : Nonce .
56  var sk : Key .
57  var AtSet : AttributeSet .
58
59  *** 1. A->B: A, Na
60  crl [rule1] :
61     < A : Agent | cnt : I , dcom : add(B,DS) ,
62        rolei : RI , AtSet >
63  =>
64     < A : Agent | cnt : (I + 1) , dcom : DS ,
65        rolei : add(run(n(A,I),B),RI) , AtSet >
66     from(A)to(B)send(A)(n(A,I))
67  if B =/= mtset .
68
69  *** 1. A->B: A, Na
70  *** 2. B->S: B,Nb,{A,Na}Kbs
71  rl [rule2] :
72     < B : Agent | cnt : I , roler : RR , AtSet >
73     *** eigentlich na, aber dann typsicher
74     from(P)to(B)send(A)(f)
75  =>
76     < B : Agent | cnt : (I + 1) , roler : add(run(n(B,I),A),RR) , AtSet >
77     from(B)to(server)send(B)(n(B,I))(ed(ServerKey(B),cat(A,f)))
78  .
79
80  *** 2. B->S: B,Nb,{A,Na}Kbs
81  *** 3. S->A: Nb,{B,Kab.Na}Kas,{A,Kab.Nb}Kbs
82  rl [rule3] :
83     from(B)to(server)send(B)(nb)(ed(ServerKey(P),cat(A,f)))
84  =>
85     if ( B =/= A )
86     then
87        from(server)to(A)send(nb)
88           (ed(ServerKey(A),cat(B,SharedKey(A,B),f)))
89           (ed(ServerKey(B),cat(A,B,SharedKey(A,B),nb)))
90     else
91        none
92     fi
93  .
94
95  *** 3. S->A: Nb,{B,Kab,Na}Kas,{A,Kab,Nb}Kbs
96  *** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
97  rl [rule4] :
98     < A : Agent | ecom : ES ,
99        rolei : add(run(na,B),RI) , AtSet >
100    from(server)to(A)send(nb)(ed(ServerKey(A),cat(P,sk,f1)))(f)
101 =>
102    if ( P == B and na == f1 )
103    then
104       < A : Agent | ecom : add(estab(i,na,B,nb,sk),ES) ,
105          rolei : RI , AtSet >
106          from(A)to(B)send(f)(ed(sk,nb))
107    else
108       < A : Agent | ecom : ES ,
109          rolei : RI , AtSet >
110    fi
111 .
```

```
112
113 *** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
114 rl [rule5] :
115    < B : Agent | ecom : ES ,
116       roler : add(run(nc,P),RR) , AtSet >
117    *** f ist der Shared-key
118    from(A)to(B)send(ed(ServerKey(B),cat(A,B,f,nb)))(ed(f,nb))
119 =>
120    if( nc == nb and P == A )
121    then
122       < B : Agent | ecom : add(estab(r,nc,A,nc,f),ES) ,
123          roler : RR , AtSet >
124    else
125       < B : Agent | ecom : ES ,
126          roler : RR , AtSet >
127    fi
128 .
129 endm
```

intruder1.maude

```
1  in model-checker .
2  mod YAHALOM-INTRUDER1 is
3  protecting YAHALOM .
4  inc MODEL-CHECKER .
5  subsort Configuration < State .
6
7  op Intruder : -> Cid .
8  *** Objekte-Attribute von Intruder
9  op agents :_ : Set  -> Attribute .
10 op attack :_ : Int -> Attribute .
11 op limit :_ : Int -> Attribute .
12 op msgs :_   : Set -> Attribute .
13 op ncs :_    : Set  -> Attribute .
14
15
16
17 vars A B P E : Principal .
18 vars AS DS ES MS NS RI RR : Set .
19 vars I LS : Nat .
20 vars f f1 : Field .
21 vars na nb nc ne : Nonce .
22 var sk : Key .
23 var Conf : Configuration .
24 var AtSet : AttributeSet .
25
26 *** Normaler Protokolllauf
27 crl [1intruder] :
28    < E : Intruder | cnt : I , dcom : add(B,DS) ,
29       rolei : RI , AtSet >
30 =>
31    < E : Intruder | cnt : (I + 1) , dcom : DS ,
32       rolei : add(run(n(E,I),B),RI) , AtSet >
33    from(E)to(B)send(E)(n(E,I))
34 if B =/= mtset .
35
36
37 rl [2intruder] :
```

```
38    < E : Intruder | agents : AS , cnt : I ,
39       roler : RR , AtSet >
40    from(A)to(E)send(A)(na)
41
42 =>
43    < E : Intruder | agents : add(A,AS) , cnt : (I + 1) ,
44       roler : add(run(n(E,I),A),RR), AtSet>
45    from(E)to(server)send(E)(n(E,I))(ed(ServerKey(E),cat(A,na)))
46 .
47
48
49 *** 3. S->A: Nb,{B,Kab,Na}Kas,{A,Kab,Nb}Kbs
50 *** 4. A->B: {A,Kab,Nb}Kbs,{Nb}Kab
51 crl [4intruder] :
52    < E : Intruder | ecom : ES ,
53       rolei : add(run(nc,P),RI) , AtSet >
54    from(server)to(E)send(nb)(ed(ServerKey(E),cat(B,sk,ne)))(f)
55 =>
56    < E : Intruder | ecom : add(estab(i,ne,B,nb,sk),ES) ,
57       rolei : RI , AtSet >
58    from(E)to(B)send(f)(ed(sk,nb))
59 if B == P and nc == ne
60 .
61
62 rl [5intruder] :
63    < E : Intruder | ecom : ES ,
64       roler : add(run(nc,P),RR) , AtSet >
65    from(A)to(E)send(ed(ServerKey(E),cat(A,sk,ne)))(ed(sk,ne))
66 =>
67    if( nc == ne and P == A )
68    then
69       < E : Intruder | ecom : add(estab(r,nc,A,mtset,sk),ES) ,
70          roler : RR , AtSet >
71    else
72       < E : Intruder | ecom : ES ,
73          roler : RR , AtSet >
74    fi
75 .
76
77 rl [intercept1] :
78    < E : Intruder | agents : AS , ncs : NS , AtSet >
79    from(A)to(B)send(A)(na)
80 =>
81    < E : Intruder | agents : add(A,B,AS) , ncs : add(na,NS) , AtSet >
82 .
83
84 rl [intercept2] :
85    < E : Intruder | agents : AS , msgs : MS , AtSet >
86    from(B)to(server)send(B)(nb)(f)
87 =>
88    < E : Intruder | agents : add(B,AS) , msgs : add(f,MS) , AtSet >
89 .
90
91
92 rl [intercept3] :
93    < E : Intruder | agents : AS , msgs : MS ,
94       ncs : NS , AtSet >
95    from(server)to(A)send(nb)(f)(f1)
96 =>
```

```
97     < E : Intruder | agents : add(A,AS) , msgs : add(f,f1,MS) ,
98       ncs : add(nb,NS) , AtSet >
99  .
100
101 rl [intercept4] :
102    < E : Intruder | agents : AS , msgs : MS , AtSet >
103    from(A)to(B)send(f)(f1)
104 =>
105    < E : Intruder | agents : add(A,B,AS) , msgs : add(f,f1,MS) , AtSet >
106 .
107
108
109 crl [replayMsg1] :
110    < E : Intruder | agents : add(A,B,AS) , limit : LS ,
111      ncs : add(nc,NS) , AtSet >
112 =>
113    < E : Intruder | agents : add(A,AS) , limit : ( LS - 1 ) ,
114      ncs : add(nc,NS) , AtSet
115    >
116    from(A)to(B)send(A)(nc)
117 if LS > 0
118 .
119
120
121 crl [replayMsg2] :
122    < E : Intruder | agents : add(A,AS) , limit : LS ,
123      msgs : add(f,MS) , ncs : add(nc,NS) , AtSet >
124 =>
125    < E : Intruder | agents : add(A,AS) , limit : ( LS - 1 ) ,
126      msgs : add(f,MS) , ncs : add(nc,NS) , AtSet >
127    from(A)to(server)send(A)(nc)(f)
128 if LS > 0
129 .
130
131 crl [replayMsg3] :
132    < E : Intruder | agents : add(A,AS) , limit : LS ,
133      msgs : add(f,f1,MS) , ncs : add(nc,NS) , AtSet >
134 =>
135    < E : Intruder | agents : add(A,AS) , limit : ( LS - 1 ) ,
136      msgs : add(f,f1,MS) , ncs : add(nc,NS) , AtSet >
137    from(server)to(A)send(nc)(f)(f1)
138 if LS > 0
139 .
140
141 crl [replayMsg4] :
142    < E : Intruder | agents : add(A,B,AS) , limit : LS ,
143      msgs : add(f,f1,MS) , AtSet >
144 =>
145    < E : Intruder | agents : add(A,B,AS) , limit : ( LS - 1 ) ,
146      msgs : add(f,f1,MS) , AtSet >
147    from(A)to(B)send(f)(f1)
148 if LS > 0
149 .
150 ***vars for attack
151 vars r1 r2 : Role .
152 vars n1 n2 n3 n4 : Nonce .
153 vars P1 P2 : Principal .
154 vars k1 k2 : Key .
155 var I1 : Int .
```

```
vars DS1 RI1 RR1 : Set .
var BI : Nat .
vars BDS BES  BRI  BRR : Set .
crl [recognizeAttack] :
   < E : Intruder | agents : AS , attack : 0 , ESET >
   < A : Agent | cnt : I1 , dcom : DS1 ,
      ecom : add(estab(i,n1,B,n2,k2),ES) , rolei : RI1 , roler : RR1 >
   < B : Agent | cnt : BI , dcom : BDS , ecom : BES ,
      rolei : BRI , roler : BRR >
=>
   < E : Intruder | agents : AS , attack : 1 , ESET >
   < A : Agent | cnt : I1 , dcom : DS1 ,
      ecom : add(estab(i,n1,B,n2,k2),ES) , rolei : RI1 , roler : RR1 >
   < B : Agent | cnt : BI , dcom : BDS , ecom : BES ,
      rolei : BRI , roler : BRR >
if  not(in(run(n2,A),BRR)) and not(in(estab(r,n2,A,n2,k2),BES))
.



*** Model-Checking
vars ASET BSET  ESET : AttributeSet .
op foundAttack : -> Prop .
eq    ((
   < E : Intruder | agents : AS , attack : 1 , cnt : I , dcom : DS , ecom : ES ,
      limit : LS , msgs : MS , ncs : NS , rolei : RI , roler : RR  >
   Conf
   ) |= foundAttack )  = true .

op init : -> Configuration .
eq init =
< 'Alice : Agent | cnt : 0 , dcom : 'Bob  , ecom : mtset ,
   rolei : mtset ,  roler : mtset >
< 'Bob   : Agent | cnt : 0 , dcom :  mtset , ecom : mtset ,
   rolei : mtset ,  roler : mtset >
< 'Eve   : Intruder | agents : mtset , attack : 0 , cnt : 0 ,
   dcom :  'Alice , ecom : mtset , limit : 6 , msgs : mtset ,
   ncs : mtset , rolei : mtset ,  roler : mtset >
.

endm
```

intruder2.maude

```
in model-checker .
mod YAHALOM-INTRUDER2 is
protecting YAHALOM .
inc MODEL-CHECKER .
subsort Configuration < State .

op Intruder : -> Cid .
*** Objekte-Attribute von Intruder
op agents :_ : Set  -> Attribute .
op attack :_ : Int -> Attribute .
op limit :_ : Int -> Attribute .
op msgs :_   : Set -> Attribute .
op ncs :_    : Set  -> Attribute .

```

```
15
16
17  vars A B P E : Principal .
18  vars AS DS ES MS NS RI RR : Set .
19  vars I LS : Nat .
20  vars f f1 : Field .
21  vars na nb nc nc1 ne : Nonce .
22  var sk : Key .
23  var Conf : Configuration .
24  var AtSet ESet : AttributeSet .
25
26  crl [intercept1] :
27     < E : Intruder | agents : AS , limit : LS , ncs : NS , ESet >
28     from(A)to(B)send(A)(na)
29  =>
30     < E : Intruder | agents : add(A,B,AS) , limit : LS , ncs : add(na,NS) , ESet >
31  if LS > 0
32  .
33
34  crl [intercept2] :
35     < E : Intruder | agents : AS , limit : LS ,
36           msgs : MS , ncs : NS , ESet >
37     from(B)to(server)send(B)(nb)(f)
38  =>
39     < E : Intruder | agents : add(B,AS) , limit : LS ,
40        msgs : add(f,MS) , ncs : add(nb,NS) , ESet >
41  if LS > 0
42  .
43
44  crl [fakeMsg1] :
45     < E : Intruder | agents : add(A,B,AS) , limit : LS ,
46        ncs : add(nc,nc1,NS) , ESet >
47  =>
48     < E : Intruder | agents : add(A,B,AS) , limit : ( LS - 1 ) ,
49        ncs : add(nc,nc1,NS) , ESet >
50     from(A)to(B)send(A)(cat(nc,nc1))
51  if LS > 0
52  .
53
54
55  crl [replayMsg1] :
56     < E : Intruder | agents : add(A,B,AS) , limit : LS ,
57        ncs : add(nc,NS) , ESet >
58  =>
59     < E : Intruder | agents : add(A,B,AS) , limit : ( LS - 1 ) ,
60        ncs : add(nc,NS) , ESet >
61     from(A)to(B)send(A)(nc)
62  if LS > 0
63  .
64
65  crl [fakeMsg4] :
66     < E : Intruder | agents : add(A,B,AS) , ecom : ES ,
67        limit : LS , msgs : add(f,f1,MS) , ncs : add(nc,nc1,NS) , ESet >
68  =>
69     < E : Intruder | agents : add(A,B,AS) , ecom : add(estab(i,nc,B,nc1,nc),ES) ,
70        limit : ( LS - 1 ) , msgs : add(f,MS) , ncs : add(nc,nc1,NS ), ESet >
71     from(A)to(B)send(f)(ed(nc,nc1))
72  if LS > 0
73  .
```

```
74
75  ***vars for attack
76  vars r1 r2 : Role .
77  vars n1 n2 n3 n4 : Nonce .
78  vars P1 P2 : Principal .
79  vars k1 k2 : Key .
80  var I1 : Int .
81  vars DS1 RI1 RR1 : Set .
82  crl [recognizeAttack] :
83     < E : Intruder | attack : 0 , ecom : add(estab(r1,n1,P1,n2,f), ES) , ESet >
84     < A : Agent | ecom : add(estab(r2,n3,P2,n4,f1),ES) , AtSet >
85  =>
86     < E : Intruder | attack : 1 , ecom : add(estab(r1,n1,P1,n2,f), ES) , ESet >
87     < A : Agent | ecom : add(estab(r2,n3,P2,n4,f1),ES) , AtSet >
88  if r1 =/= r2 and P1 == A and P2 =/= E and f == f1
89  .
90  *** Model-Checking
91  op foundAttack : -> Prop .
92  eq    ((
93     < E : Intruder | agents : AS , attack : 1 , cnt : I , dcom : DS ,
94        ecom : ES , limit : LS , msgs : MS , ncs : NS , rolei : RI , roler : RR
95     >
96     Conf
97     ) |= foundAttack )  = true .
98
99  op init : ->  Configuration .
100 eq init =
101 < 'Alice : Agent | cnt : 0 , dcom : 'Bob  , ecom : mtset ,
102    rolei : mtset ,  roler : mtset >
103 < 'Bob   : Agent | cnt : 0 , dcom : mtset  , ecom : mtset ,
104    rolei : mtset ,  roler : mtset >
105 < 'Eve   : Intruder | agents : add('Alice,'Bob) , attack : 0 , cnt : 0 ,
106    dcom : mtset , ecom : mtset , limit : 3 , msgs : mtset ,
107    ncs : mtset , rolei : mtset ,  roler : mtset >
108 .
109
110 endm
111 set trace on .
```

Literaturverzeichnis

[BAN89] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, February 1989.

[Böh02] Thomas Böhne. A general intruder for security protocols in Maude. Master's thesis, University of Wyoming, aug 2002.

[CDE+03] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *The Maude 2.0 Manual*, June 2003.

[EC99] D. Peled E. Clarke, O. Grumberg. *Model Checking*. MIT, Cambridge, Mass., 1999.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[Syv94] Paul Syverson. A taxonomy of replay attacks. In *Computer Security Foundations Workshop VII*, pages 131–136. IEEE Computer Society Press, 1994.