Model Checking mit SPIN – Beispiele

Eine Überraschung für Mordechai Ben-Ari

In seinem Buch „Principles of the Spin Model Checker“ [1] beschreibt Mordechai Ben-Ari ein simples Szenario zweier nebenläufiger Programme, die eine gemeinsame Ressource verwenden. Es scheint so zu sein, dass man leicht versteht, was bei der Ausführung alles passieren kann. Oder doch nicht?

Einfaches Beispiel von Wechselwirkung zwischen Prozessen

Der Modelchecker SPIN hat die Sprache PROMELA[1] in der die Modelle definiert werden, deren Eigenschaften überprüft werden sollen.

Das folgende Listing `il1.pml` zeigt ein einfaches solches Programm:

```
1  #include "for.h"
2  byte n = 0;
3
4  proctype P() {
5    byte temp;
6    for (i, 1, 10)
7      temp = n + 1;
8      n = temp
9    rof(i)
10 }
11
12 init {
13   atomic {
14     run P();
15     run P();
16   }
17   (_nr_pr == 1) ->
18     printf("The value is %d\n", n);
19 }
```

PROMELA hat numerische Datentypen, von denen `byte` einer ist. In Zeile 2 wird ein Byte `n` mit `0` initialisiert.

Programme in PROMELA bestehen aus *Prozessen*, die als `proctype` definiert werden. In den Zeilen 4 - 10 wird ein Prozess `P` definiert, der eine lokale Variable `temp` verwendet. In der Include-Datei `for.h` hat Ben-Ari ein Makro definiert, das eine `for`-Schleife simuliert. PROMELA selbst hat nur `do .. od` um Schleifen zu spezifizieren, `for.h` verwendet `do`. In der `for`-Schleife liest `P` die globale Variable `n` und schreibt ihr Inkrement in die lokale Variable

[1] PROMELA steht für *Process* oder *Protocol Meta Language*

`temp`. Danach wird der globalen Variablen `n` der inkrementierte Wert in `temp` zugewiesen.[2]

Eine spezielle Rolle spielt der Prozess `init`, der in den Zeilen 12 - 19 definiert wird. Hat ein Programm einen solchen Prozess, dann wird er stets zuerst aktiviert und bekommt die Prozess-Id `_pid 0`.

In unserem Beispiel startet `init` zwei Instanzen des Prozesses `P`. `atomic` sorgt dafür, dass beide Prozesse zunächst erzeugt werden und nicht der erste schon startet, ehe der zweite erzeugt ist.

In Zeile 17 wird die eingebaute Variable `_nr_pr` überprüft. Sie enthält stets die aktuelle Zahl der Prozesse. Ein Ausdruck wie `(_nr_pr == 1)`, der zu einem Wahrheitswert auswertet, ist in PROMELA genau dann *ausführbar*, wenn er `true` ist. Das bedeutet, dass in Zeile 17 der Prozess `init` solange blockiert, bis die Bedingung wahr wird, d.h. die Zahl der Prozesse `1` wird, die beiden gestarteten Prozesse `P` also beendet sind. Tritt das ein, wird der Wert der globalen Variablen `n` ausgegeben.

Es ist recht einfach, einen Ablauf des Programms zu machen:

```
spin il1.pml
```

Mit diesem Kommando erzeugt SPIN *einen* möglichen Ablauf, das Programm wird *simuliert*.

Die Ausgabe eines solchen Ablaufs:

```
   The value is 17
3 processes created
```

Wiederholt man die Simulation, wird SPIN einen anderen Ablauf erzeugen und ein anderes Ergebnis zeigen.

Macht man das öfters, wird man feststellen, dass die Werte zwischen `10` und `20` liegen. Schaut man sich die denkbaren Abläufe genauer an, sieht man, dass dies kein Zufall ist.

Denkbare Abläufe

Die beiden Prozesse, nennen wir sie P_1 und P_2, werden nebenläufig durchgeführt. Dadurch kann es vorkommen, dass die Aktionen der beiden Prozesse verschränkt ausgeführt werden.

Man kann sich zwei gewissermaßen extreme Ausprägungen dieser Verschränkung vorstellen, nämlich gar keine oder die „maximale“, bei der die Teilschritte der beiden Prozesse immer abwechselnd durchgeführt werden.

[2] Es gibt in PROMELA auch den Inkrement-Operator `++`. In unserem Beispiel wollen wir ihn jedoch nicht verwenden, sondern davon ausgehen, dass wir eine Maschine modellieren, in der das Inkrement dadurch gebildet wird, dass eine Wert aus einer Speicherzelle nur dadurch verändert werden kann, dass er in ein Register geladen wird und dann nach der Änderung in die Speicherzelle zurückgeschrieben wird. `n` wäre also die Speicherzelle und `temp` das Register.

Ablauf 1: Die beiden Prozesse werden *sequenziell* ausgeführt, sage zuerst P_1, dann P_2.

Zuerst wird also P_1 die globale Variable schrittweise auf 10 erhöhen und danach macht P_2 weiter und inkrementiert auch 10 mal.

Das Ergebnis: `The value is 20`

Ablauf 2: Die beiden Prozesse sind verschränkt und zwar so, dass ihre Teilschritte immer *abwechselnd* aufeinander folgen.

Zuerst setzt also P_1 seine lokale Variable auf `1`, dann P_2 die seinige ebenfalls auf `1`. Dann weist P_1 der globalen Variablen `n` die `1` zu und schließlich macht auch P_2 diese Anweisung `n = 1`.

Jedes Paar von Schleifendurchläufen der beiden Prozesse erhöht also die globale Variable `n` um `1` und folglich hat sie am Ende den Wert `10`.

Das Ergebnis: `The value is 10`

Also liegt die Schlussfolgerung nahe, dass unsere Überlegungen belegen, dass für jeden denkbaren verschränkten Ablauf der beiden Prozesse am Ende gilt: `10 <= n <= 20`. Wirklich?

Die Überraschung

Ein Student von Ben-Ari hat immer wieder die Simulation in SPIN aufgerufen. Er war offenbar ziemlich ausdauernd – und plötzlich verblüfft: Das Ergebnis war bei einem Durchlauf `9`! [1, S.42]

Und es stellt sich sogar heraus, dass die Variable `n` am Ende sogar den Wert `2` haben kann.

"Intuitively, it seems as if 'perfect' interleaving represents the maximum 'amount of interference' possible, and for many years I taught that the final value of `n` must be between `10` and `20`. It came somewhat of a shock when I discovered that the final value can be as low as `2`!" [1, S.39f.]

Besser also, man überprüft die Vermutung. Und das kann man mit SPIN tun.

Modelchecking an diesem Beispiel

Eine einfache Art, SPIN zu verwenden, besteht darin, dass man bestimmte Bedingungen annimmt und SPIN überprüfen lässt, ob Gegenbeispiele existieren.

In unserem Beispiel wollen wir prüfen, ob als Ergebnis auch eine Wert herauskommen kann, der kleiner als `10` ist, ja sogar, ob als Ergebnis der Wert `2` auftreten kann.

Dazu hat SPIN das Konstrukt `assert`. Es bedeutet, dass SPIN ein Gegenbeispiel erzeugt, falls es *irgendeinen* Durchlauf gibt, bei dem die Bedingung des `assert` *nicht* erfüllt ist.

Wollen wir also prüfen, ob es einen Durchlauf mit dem Ergebnis 2 gibt, nehmen wir an, dass das nicht der Fall ist und lassen uns von Spin ein Gegenbeispiel produzieren, also eines das 2 als Ergebnis hat. Tun wir das in der Datei `il2.pml`:

```
1  #include "for.h"
2  byte n = 0;
3
4  proctype P() {
5    byte temp;
6    for (i, 1, 10)
7      temp = n + 1;
8      n = temp
9    rof(i)
10 }
11
12 init {
13   atomic {
14     run P();
15     run P();
16   }
17   (_nr_pr == 1) ->
18     printf("The value is %d\n", n);
19     assert(n > 2)
20 }
```

Die einzige Veränderung ist die neue Zeile 19, bei der nach Ende der beiden Prozesse fordern, dass `(n > 2)` gilt. Wir lassen Spin dann prüfen, ob das wirklich der Fall ist.

Dazu geht man so vor:

1. Mit dem Kommando `spin -a il2.pml` erzeugt Spin C-Code für den *Verifier*, der dann die eigentliche Überpüfung durchführt.

   Ergebnis der Ausführung des Kommandos sind einige C-Quellen, insbesondere `pan.c` und `pan.h`.

2. Wir übersetzen diese Quellen mit `gcc -DSAFETY -o pan pan.c` (oder `-o pan.exe` unter Windows) Dabei verwenden wir das *define* `SAFETY`, weil wir nur eine Sicherheitsbedingung prüfen wollen. [3]

3. Nun kann der *Verifier* `pan` (*process analyzer*) ausgeführt werden und wir erhalten tatsächlich eine Fehlermeldung.

   ```
   pan:1: assertion violated (n>2) (at depth 90)
   pan: wrote il2.pml.trail
   ```

   Die dabei erzeugte Datei `il2.pml.trail` enthält den „Zeugen“, einen Durchlauf, bei dem die gewünschte Bedingung verletzt wurde.

[3] Spin kann in vielfacher Hinsicht parametrisiert werden, um möglichst effizienten C-Code zu produzieren. Siehe http://spinroot.com/spin/Man/Pan.html

4. Die geführte Simulation `spin -t -p -g il2.pml` verwendet diesen Zeugen und gibt das Szenario aus, das zur Verletzung der Bedingung geführt hat – in unserem Beispiel einen Ablauf, der als Ergebnis den Wert 2 hat.

Die folgende Datei zeigt diesen Ablauf und es wird jetzt auch ersichtlich, wie dieses überraschende Ergebnis entstanden ist.

```
Starting P with pid 1
  1:  proc  0 (:init:) il2.pml:21 (state 1) [(run P())]
Starting P with pid 2
  2:  proc  0 (:init:) il2.pml:22 (state 2) [(run P())]
  3:  proc  2 (P) il2.pml:13 (state 1)  [i = 1]
  4:  proc  2 (P) il2.pml:13 (state 4)  [else]
  5:  proc  1 (P) il2.pml:13 (state 1)  [i = 1]
  6:  proc  1 (P) il2.pml:13 (state 4)  [else]
  7:  proc  2 (P) il2.pml:14 (state 5)  [temp = (n+1)]
  8:  proc  1 (P) il2.pml:14 (state 5)  [temp = (n+1)]
  9:  proc  2 (P) il2.pml:15 (state 6)  [n = temp] n = 1
  ...
 13:  proc  2 (P) il2.pml:15 (state 6)  [n = temp] n = 2
  ...
  ...
 37:  proc  2 (P) il2.pml:15 (state 6)  [n = temp] n = 8
  ...
 41:  proc  2 (P) il2.pml:15 (state 6)  [n = temp] n = 9
 42:  proc  2 (P) il2.pml:16 (state 7)  [i = (i+1)]
 43:  proc  2 (P) il2.pml:13 (state 4)  [else]
 44:  proc  1 (P) il2.pml:15 (state 6)  [n = temp] n = 1
 45:  proc  1 (P) il2.pml:16 (state 7)  [i = (i+1)]
 46:  proc  1 (P) il2.pml:13 (state 4)  [else]
 47:  proc  2 (P) il2.pml:14 (state 5)  [temp = (n+1)]
 48:  proc  1 (P) il2.pml:14 (state 5)  [temp = (n+1)]
 49:  proc  1 (P) il2.pml:15 (state 6)  [n = temp] n = 2
  ...
  ...
 73:  proc  1 (P) il2.pml:15 (state 6)  [n = temp] n = 8
  ...
 77:  proc  1 (P) il2.pml:15 (state 6)  [n = temp] n = 9
  ...
 81:  proc  1 (P) il2.pml:15 (state 6)  [n = temp] n = 10
 82:  proc  1 (P) il2.pml:16 (state 7)  [i = (i+1)]
 83:  proc  1 (P) il2.pml:13 (state 2)  [((i>10))]
 84:  proc  2 (P) il2.pml:15 (state 6)  [n = temp] n = 2
 85:  proc  2 (P) il2.pml:16 (state 7)  [i = (i+1)]
 86:  proc  2 (P) il2.pml:13 (state 2)  [((i>10))]
 87: proc 2 terminates
 88: proc 1 terminates
 89:  proc  0 (:init:) il2.pml:24 (state 4) [((nr_pr==1))]
      The value is 2
 90:  proc  0 (:init:) il2.pml:25 (state 5) [printf('The value is %d\\n',n)]
```

```
spin: il2.pml:26, Error: assertion violated
spin: text of failed assertion: assert((n>2))
 91:  proc  0 (:init:) il2.pml:26 (state 6) [assert((n>2))]
spin: trail ends after 91 steps
#processes: 1
    n = 2
 91:  proc  0 (:init:) il2.pml:28 (state 7) <valid end state>
3 processes created
```

In Schritt 8 speichert P_1 `1` in seiner lokalen Variablen `temp`. Ab dann läuft erstmal nur P_2 und erhöht `n` schließlich auf `9` in Schritt 41.

In Schritt 44 schreibt nun P_1 seinen lokalen Wert von `1` in die Variable `n`.

In Schritt 47 liest P_2 die globale Variable und merkt sich `2` in seiner lokalen Variablen `temp`.

Ab jetzt ist nur P_1 dran, verwendet seinen lokalen Wert von `1` und erhöht sukzessive bis `10` in Schritt 81. Jetzt ist P_1 fertig, aber P_2 muss noch den 10. Schleifendurchgang beenden und schreibt den in Schritt 47 gemerkten Wert von `2` in die globale Variable `n`.

So ist es passiert!

Überprüfung kryptographischer Protokolle

Das Needham-Schroeder-Protokoll (1978) dient der wechselseitigen Authentifizierung zweier Partner. Das Protokoll ist im Grunde eine Kombination zweier Abläufe: (1) dem Beziehen von öffentlichen Schlüsseln von einem Authentifizierungsserver und (2) der eigentlichen wechselseitigen Authentifizierung der Partner. Es ist schon lange bekannt, dass das Protokoll unsicher ist, weil es nicht garantiert, dass die bezogenen öffentlichen Schlüssel aktuell und nicht kompromittiert sind. Dieses Problem lässt sich jedoch leicht durch Zeitstempel verhindern. Das Protokoll wird dargestellt in [4, S. 69ff].

Erst 1995 hat Gavin Lowe [2] gezeigt, dass auch der Teil zur wechselseitigen Authentifizierung fehlerhaft ist. Diesen Nachweis kann man durch Model Checking führen.

Das Needham-Schroeder-Protokoll

Wir betrachten nur den Teil des Needham-Schröder-Protokolls zur wechselseitigen Authentifizierung und unterstellen, dass die beiden Partner, Alice und Bob[4] die öffentlichen Schlüssel des jeweils anderen Partners bereits haben.

Sei im Folgenden K_A der öffentliche Schlüssel von Alice und K_B der von Bob. Mit $enc_K(M)$ bezeichnen wir die Verschlüsselung der Nachricht M mit dem Schlüssel K.

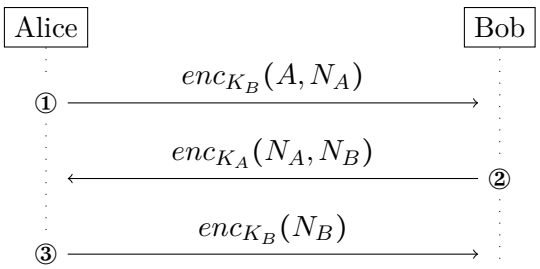

Abbildung 1: Authentifizierung nach Needham-Schroeder

Dieser Teil des Needham-Schroeder-Protokolls besteht dann aus drei Schritten (siehe 1):

① Alice erzeugt eine Zufallszahl (*nonce*) N_A und sendet $enc_{K_B}(A, N_A)$, wobei A eine Nachricht ist, die die Identität von Alice enthält. Bob entschlüsselt diese Nachricht und kennt nun das Geheimnis N_A von Alice.

② Bob erzeugt auch eine Zufallszahl N_B und sendet $enc_{K_A}(N_A, N_B)$ zurück. Alice entschlüsselt die Nachricht und kennt nun das Geheimnis

[4] In der Beschreibung kryptographischer Protokolle wird üblicherweise die einleitende Beteiligte *Alice* und der andere Partner *Bob* genannt.

N_B von Bob. Da Alice ihr eigenes Geheimnis von Bob zurückerhält, ist sie überzeugt, mit Bob zu kommunizieren.

③ Alice sendet nun die von Bob erhaltene Zufallszahl N_B in $enc_{K_B}(N_B)$ zurück. Bob entschlüsselt die Nachricht. Er erhält seine eigene Zufallszahl zurück und ist überzeugt, mit Alice zu kommunizieren.

Ein Angreifer (*intruder*) kann wie jeder andere Teilnehmer innerhalb eines Netzwerkes als Partner in einer Kommunikation auftreten. Er kann aber nur solche Nachrichten entschlüsseln, die mit seinem eigenen Schlüssel verschlüsselt wurden. Er kann natürlich Tricks versuchen, z.B. sich als eine andere Person ausgeben, oder Nachrichten abfangen und später verschicken u.ä.

Modellierung in PROMELA

Stephan Merz [3] http://www.loria.fr/~merz/ hat das Protokoll sowie die Aktionen eines potentiellen Angreifers in PROMELA, der Sprache des Modelcheckers SPIN modelliert. Man kann dann mit SPIN zeigen, dass das Protokoll unterlaufen werden kann.

Das Modell des Protokolls macht einige vereinfachende Annahmen:

- Es gibt drei Akteure: Bob, Alice und I, den Intruder.
- Alice initiert den Ablauf mit Bob oder I.
- Bob tritt nur als Antwortender auf.
- Es gibt nur einen Lauf für Alice und Bob.
- Der Intruder kann Nachrichten auf dem Netzwerk lesen.

Merz definiert zunächst einige Typen, die die Lesbarkeit der Spezifikation erleichtern:

```
mtype = { ok, err, msg1, msg2, msg3, keyA, keyB, keyI,
          agentA, agentB, agentI, nonceA, nonceB, nonceI };
```

Nachrichten bestehen aus einem Schlüssel (dem öffentlichen Schlüssel des intendierten Empfängers) und zwei Datenteilen:

```
typedef Crypt { mtype key, data1, data2 };
```

Das Netzwerk, auf dem die Kommunikation abgewickelt wird, wird modelliert als Kanal (*channel*), den alle Akteure verwenden. In PROMELA wird synchrone Kommunikation durch einen Kanal der Puffergröße 0 dargestellt. Das bedeutet, dass der Sender warten muss, wenn der Empfänger nicht bereit ist und umgekehrt.

```
chan network = [0] of { mtype, /* msg# */
                        mtype, /* receiver */
                        Crypt  /* encrypted msg */};
```

Alice wird nun als Prozess modelliert. In PROMELA wird in einer `if`-Anweisung eine aus eventuell mehreren ausführbaren Anweisungen (nichtdeterministisch) genommen. Im Prozess für Alice wählt Alice somit in der `if`-Anweisung Bob oder den Intruder als Partner der Kommunikation.

Alice verschickt dann ihre erste Nachricht und wartet auf die Antwort. Hat die Antwort ihren Schlüssel und enthält sie als ersten Teil ihre Zufallszahl, liest Alice das Geheimnis ihres Partners. Dann verschickt sie ihre Antwort. Kommt der Prozess zu dieser Stelle, ist der Status für Alice in Ordnung.

```
mtype partnerA;
mtype statusA = err;
active proctype Alice() {
  mtype pkey, pnonce;
  Crypt data;

  if /* choose a partner for this run */
  :: partnerA = agentB; pkey = keyB;
  :: partnerA = agentI; pkey = keyI;
  fi;
  network ! (msg1, partnerA, Crypt{pkey, agentA, nonceA});

  network ? (msg2, agentA, data);
  (data.key == keyA) && (data.info1 == nonceA);
  pnonce = data.info2;

  network ! (msg3, partnerA, Crypt{pkey, pnonce, 0});
  statusA = ok;
}
```

Bob wird ganz analog modelliert, siehe Anhang.

Der Intruder versucht nun Nachrichten, die er empfängt zu entschlüsseln und weiterzuleiten oder, wenn er sie nicht entschlüsseln kann, wie empfangen weiterzuleiten.

```
bool knows_nonceA, knows_nonceB;

active proctype Intruder() {
  mtype msg, recpt;
  Crypt data, intercepted;

  do
  :: network ? (msg, _, data) ->
    if /* perhaps store the message */
    :: intercepted = data;
    :: skip;
    fi;
    if /* record newly learnt nonces */
    :: (data.key == keyI) ->
      if
```

```
        :: (data.info1 == nonceA) || (data.info2 == nonceA)
        -> knows_nonceA = true;
        :: else -> skip;
        fi;
        /* similar for knows_nonceB */
      :: else -> skip;
      fi;
    :: /* Replay or send a message */
      if /* choose message type */
      :: msg = msg1;
      :: msg = msg2;
      :: msg = msg3;
      fi;
      if /* choose recipient */
      :: recpt = agentA;
      :: recpt = agentB;
      fi;
      if /* replay intercepted message or assemble it */
      :: data = intercepted;
      :: if
        :: data.info1 = agentA;
        :: data.info1 = agentB;
        :: data.info1 = agentI;
        :: knows_nonceA -> data.info1 = nonceA;
        :: knows_nonceB -> data.info1 = nonceB;
        :: data.info1 = nonceI;
        fi;
        /* similar for data.info2 and data.key */
      fi;
      network ! (msg, recpt, data);
    od;
  }
```

Verfikation mit SPIN

Zur Verifikation müssen wir nun die überprüfende Eigenschaft als LTL-Formel ausdrücken:

Seien $statusA, statusB$ die Flags, die anzeigen, ob der Lauf aus Sicht von Alice bzw. Bob erfolgreich durchlaufen wurde. $partnerA, partnerB$ seien, die Partner, die ihre Geheimnisse ausgetauscht haben und $knows\_nonceA$, $knows\_nonceB$ gibt an, ob der Intruder das Gehemins von Alice bzw. Bob kennt.

Dann kann man die gewünschte Eigenschaft des Protokoll so ausdrücken:

$$\Box\,(statusA = ok \wedge statusB = ok \rightarrow (partnerA = agentB \rightarrow partnerB = agentA))$$

$$\Box\,(statusA = ok \wedge statusB = ok \rightarrow (partnerB = agentA \rightarrow partnerA = agentB))$$

$$\Box\,(statusA = ok \wedge partnerA = agentB \rightarrow \neg knows\_nonceA)$$

$$\Box\,(statusB = ok \wedge partnerB = agentA \rightarrow \neg knows\_nonceB)$$

Diese Aussagen übernehmen wir in die PROMELA-Datei:

```
#define success statusA == ok && statusB == ok
#define aliceBob  partnerA == bob
#define bobAlice  partnerB == alice
#define nonceASecret  !knowNA
#define nonceBSecret  !knowNB

/* Verification formulae in LTL */
ltl fml1 { []  (success -> (aliceBob -> bobAlice)) }
ltl fml2 { []  (success -> (bobAlice -> aliceBob)) }
ltl fml3 { []  ((aliceBob && success) -> nonceASecret) }
ltl fml4 { []  ((bobAlice && success) -> nonceBSecret) }
```

Wir können nun die einzelnen Aussagen checken:

```
spin -a NeedhamSchroeder.pml
gcc -o pan pan.c
pan -N fml2 NeedhamSchroeder.pml
```

Mit der zweiten Aussage `fml2` zeigt der von SPIN erzeugte *Verifier pan* (*process analyzer*) einen Fehler an.

Analyse des Ergebnisses

Wenn bei der Verifikation einer Spezifikation gegenüber einer LTL-Formel ein Fehler auftritt, fertigt SPIN eine sogenannte *Trail*-Datei, die einen Pfad zeigt, auf dem die Formel verletzt wird. Diese Datei kann man für die Analyse des Ergebnisses verwenden. In unserem Fall ist es am einfachsten mit

```
spin -t -c NeedhamSchroeder.pml  // Ausgabe auf Konsole
spin -t -M NeedhamSchroeder.pml  // Ausgabe als Tcl/Tk-Datei
                                 // und Anzeige mit wish
```

eine Ausgabe als MSC (*Message Sequence Chart* zu erzeugen und diese zu analysieren. Unsere Analyse hat Abb. 2 zum Resultat.

Wenn man dieses Diagramm genau durchsieht, erhält man das in Abb. 3 dargestellte Szenario.

① Alice möchte eine Sitzung mit I eröffnen und sendet deshalb ihre Identität und ihr Geheimnis. I kann diese Nachricht entschlüsseln und kennt nun das Geheimnis von Alice.

② I sendet die Nachricht von Alice an Bob weiter, verschlüsselt mit dessen Schlüssel. I spiegelt vor, Alice zu sein.

③ Bob denkt, mit Alice zu kommunizieren und sendet nun sein Geheimnis, das I auf dem Netzwerk abfängt.

④ Diese Nachricht von Bob kann I zwar nicht entschlüsseln, er leitet sie aber an Alice weiter.

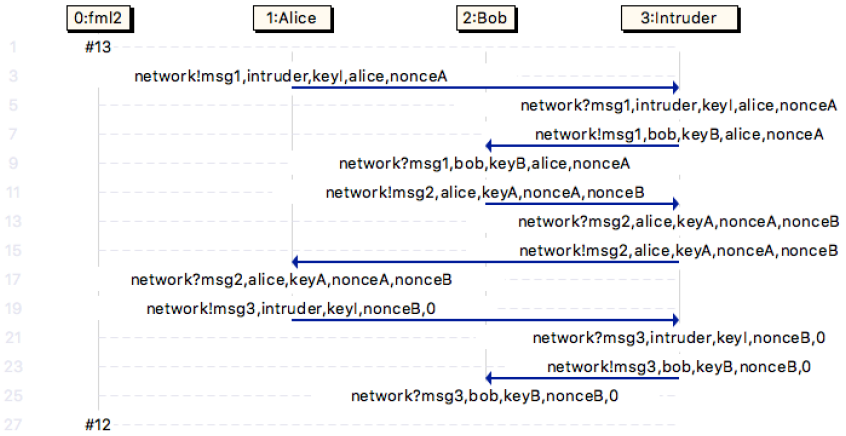

Abbildung 2:

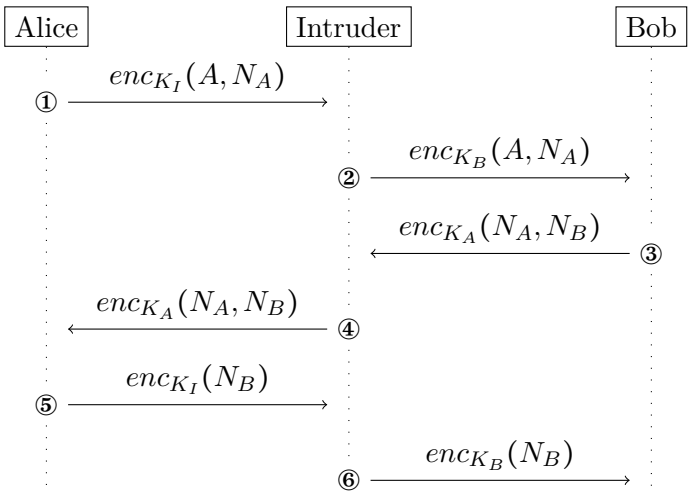

Abbildung 3: Angriffsszenario zum Needham-Schroeder-Protokoll

Alice sieht dies als die Antwort auf ihre eigene Nachricht an und entschlüsselt sie. Sie kennt nun das Geheimnis von Bob, hält dieses jedoch für das Geheimnis von I.

⑤ Alice sendet nun das vermeintliche Geheimnis von I (tatsächlich das von Bob) an I zurück. I kann nun die Nachricht entschlüsseln und kennt nun auch das Geheimnis von Bob.

⑥ I sendet die Nachricht von Alice verschlüsselt mit Bobs Schlüssel an Bob weiter. Bob denkt, mit Alice zu sprechen!

Das Ergebnis ist somit:

- I, der *Intruder*, kennt die Geheimnisse von Alice und Bob.
- Alice denkt (korrekterweise) eine Sitzung mit I initiiert zu haben.
- Bob denkt mit Alice eine Sitzung initiiert zu haben, spricht jedoch tatsächlich mit I. Der Intruder hat die Identität von Alice übernommen.

Literaturverzeichnis

[1] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. London, 2008.

[2] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 55(3):131–133, 1995.

[3] S. Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M.D.Ryan, editors, *Modeling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 3–38. Springer-Verlag, 2001.

[4] Bruce Schneier. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C*. Bonn, 1996.

Anhang: PROMELA-Spezifikation von Needham-Schroeder

```
/* (c) Copyright 2001 by Stephan Merz */

/* The Needham-Schroeder public-key protocol, as a Promela model */

/* There are three agents: Alice, Bob, and Intruder, who communicate
   via a common network. Initially, each agent has a (non-corrupted)
   pair of keys and has made up a nonce.

   Agents Alice and Bob are the initiator and responder, respectively,
   who try to establish a common secret, represented by their pair of
   nonces. The goal of the protocol is to try and convince each other
   of their presence and identity. The intruder may participate in runs
   of the protocol just as any other agent, but may also fake
   messages, using information intercepted from network traffic.
   However, even Intruder cannot break the cryptographic algorithm,
   that is, he cannot decipher messages encrypted with a key other
   than his own.

   The Needham-Schroeder protocol between agents A and B is as
   follows:

     1: A -> B: {N(A), A}_B
     2: B -> A: {N(A), N(B)}_A
     3: A -> B: {N(B)}_B

   Here, N(A) and N(B) denote the nonces of agents A and B, and
   {M}_A represents the message M encrypted with the public key of A
   (so that it can be decrypted only by A, using its private key).

   Note: This simplified version of the protocol assumes public keys
   to be known to all clients. The original version includes messages
   to request keys from a key server.
*/

mtype = {msg1, msg2, msg3, alice, bob, intruder,
         nonceA, nonceB, nonceI, keyA, keyB, keyI, ok};

typedef Crypt { /* the encrypted part of a message */
  mtype key, d1, d2;
}

/* A message in transit is modelled as a tuple
     msg# ( receiver, encrypted_data )
   The receiver field identifies the intended recipient, although an
   attacker may intercept any message sent on the network.
*/
chan network = [0] of {mtype, /* msg# */
                       mtype, /* receiver */
```

```
                    Crypt};

/* The partners successfully identified (if any) by initiator
   and responder, used in correctness assertion.
*/
mtype partnerA, partnerB;
mtype statusA, statusB;

/* Knowledge about nonces gained by the intruder. */
bool knowNA, knowNB;

active proctype Alice() { /* honest initiator for one protocol run */
  mtype  partner_key, partner_nonce;
  Crypt  data;

  if /* nondeterministically choose a partner for this run */
  :: partnerA = bob; partner_key = keyB;
  :: partnerA = intruder; partner_key = keyI;
  fi;

  d_step {
    /* Construct msg1 and send it to chosen partner */
    data.key = partner_key;
    data.d1 = alice;
    data.d2 = nonceA;
  }
  network ! msg1(partnerA, data);

  /* wait for partner to send back msg2 and decipher it */
  network ? msg2(alice, data);
  end_errA:
    /* make sure the partner used A's key and that the first
       nonce matches, otherwise block. */
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;

  d_step {
    /* respond with msg3 and declare success */
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = 0;
  }
  network ! msg3(partnerA, data);
  statusA = ok;
} /* proctype Alice() */

active proctype Bob() { /* honest responder for one run */
  mtype partner_key, partner_nonce;
  Crypt data;
```

```
  /* wait for msg1, identifying partner */
  network ? msg1(bob, data);
  /* try to decipher the message; block on wrong key */
  end_errB1:
  (data.key == keyB);
  partnerB = data.d1;

  d_step {
    partner_nonce = data.d2;
    /* lookup the partner's public key */
    if
    :: (partnerB == alice) -> partner_key = keyA;
    :: (partnerB == bob) -> partner_key = keyB;  /* shouldn't happen */
    :: (partnerB == intruder) -> partner_key = keyI;
    fi;
    /* respond with msg2 */
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = nonceB;
  }
  network ! msg2(partnerB, data);

  /* wait for msg3, check the key and the nonce, and declare success */
  network ? msg3(bob, data);
  end_errB2:
  (data.key == keyB) && (data.d1 == nonceB);
  statusB = ok;
}

active proctype Intruder() {
  /* The intruder doesn't follow a fixed protocol (we want the
     modelchecker to find the attack!), we simply list the different
     actions it can perform.
  */
  mtype msg;
  Crypt data, intercepted;
  mtype icp_type;   /* type of intercepted message */

  do
  :: /* Send msg1 to B (sending it to anybody else would be foolish).
        May use own identity or pretend to be A; send some nonce known to I.
     */
     if /* either replay intercepted message or construct a fresh message */
     :: icp_type == msg1 -> network ! msg1(bob, intercepted);
     :: data.key = keyB;
  if
  :: data.d1 = alice;
  :: data.d1 = intruder;
  fi;
  if
```

```
:: knowNA -> data.d2 = nonceA;
:: knowNB -> data.d2 = nonceB;
:: data.d2 = nonceI;
fi;
      network ! msg1(bob, data);
   fi;
:: /* Send msg2 to A. */
   if
   :: icp_type == msg2 -> network ! msg2(alice, intercepted);
   :: data.key = keyA;
      if
:: knowNA -> data.d1 = nonceA;
:: knowNB -> data.d1 = nonceB;
:: data.d1 = nonceI;
fi;
if
:: knowNA -> data.d2 = nonceA;
:: knowNB -> data.d2 = nonceB;
:: data.d2 = nonceI;
fi;
      network ! msg2(alice, data);
   fi;
:: /* Send msg3 to B. */
   if
   :: icp_type == msg2 -> network ! msg3(bob, intercepted);
   :: data.key = keyB;
if
:: knowNA -> data.d1 = nonceA;
:: knowNB -> data.d1 = nonceB;
:: data.d1 = nonceI;
fi;
      data.d2 = 0;
      network ! msg3(bob, data);
   fi;
:: /* Receive or intercept a message from A or B. If possible, extract nonces. */
   network ? msg (_, data) ->
   if /* Perhaps store the data field for later use */
   :: d_step {
  intercepted.key = data.key;
  intercepted.d1 = data.d1;
  intercepted.d2 = data.d2;
        icp_type = msg;
}
   :: skip;
   fi;
   d_step {
if /* Try to decrypt the message if possible */
:: (data.key == keyI) -> /* Have we learnt a new nonce? */
   if
   :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA = true;
```

```
            :: else -> skip;
      fi;
      if
      :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB = true;
      :: else -> skip;
      fi;
  :: else -> skip;
  fi;
     }
  od;
}

#define success statusA == ok && statusB == ok
#define aliceBob  partnerA == bob
#define bobAlice  partnerB == alice
#define nonceASecret  !knowNA
#define nonceBSecret  !knowNB

/* Verification formulae in LTL */
ltl fml1 { []  (success -> (aliceBob -> bobAlice)) }
ltl fml2 { []  (success -> (bobAlice -> aliceBob)) }
ltl fml3 { []  ((aliceBob && success) -> nonceASecret) }
ltl fml4 { []  ((bobAlice && success) -> nonceBSecret) }
```