Erstes Lab zur Vorlesung „Formale Methoden im Software Entwurf“

TECHNISCHE UNIVERSITÄT DARMSTADT

Modellierung mit PROMELA

Formale Anforderungen an Ihre Abgabe

Abgabefrist

Abgabedeadline des Labs: Mo, 10.6.2024, 8:00

Abgabeort: https://moodle.informatik.tu-darmstadt.de/mod/assign/view.php?id=67787

Besprechung der Lösung des Labs: Mo, 10.6.2024, 8:55

Sie können Ihre Abgabe bis zur Abgabefrist beliebig häufig aktualisieren. Die zuletzt fristgerecht hochgeladene Version wird bewertet. Es reicht, wenn eine Person Ihrer Labgruppe die Lösung hochlädt.

Bitte prüfen Sie, dass Ihre Abgabe aktuell und vollständig ist. Verspätete Abgaben können nicht akzeptiert werden, da die Lösung des Labs direkt nach der Abgabefrist besprochen wird.

Abgabeformat

Die Lösung ist als ein einziges zip-Archiv mit dem Namen: `Lab1_GruppeXYZ.zip` abzugeben, wobei Sie bitte `XYZ` mit Ihrer Labgruppennummer ersetzen.

Das Archiv muss die Unterverzeichnisse `aufgabe1`, `aufgabe2` und `aufgabe3` mit den Lösungsdateien der zugehörigen Aufgaben enthalten. Die Dateien müssen alle Informationen für die jeweilige Lösung beinhalten.

Nicht-Einhaltung der Ordnerstruktur oder das Fehlen von Dateien für Aufgaben, kann zu Punktabzügen führen.

Die abzugebenden PROMELA-Modelle dürfen keine Syntaxfehler beinhalten.

Weitere Bearbeitungshinweise

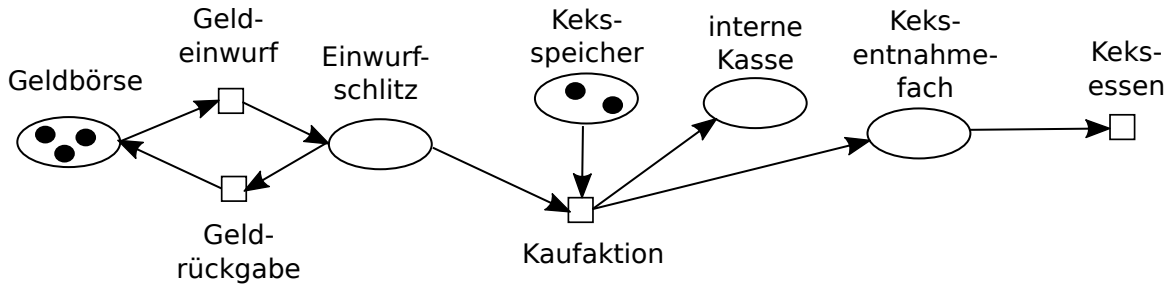
- Sie können maximal **60 Punkte** erreichen.
- Sehen Sie sich die Ergebnisse von Spin im Detail an und akzeptieren Sie nicht unkritisch die gegebene Antwort.
- Eigenschaften sind in diesem Lab ausschließlich mit Assertions und gegebenenfalls Ghostvariablen (unter Umständen in separaten Prozessen) zu spezifizieren. **LTL Formeln (siehe Vorlesung am 27.5.2024) dürfen nicht verwendet werden**.
- Prüfen Sie bei der Verifikation, dass Spin den gesamten Zustandsraum durchsucht hat und nicht wegen Speichermangel „`reached -DMEMLIM bound`“ oder zu kleiner Suchtiefe „`max search depth too small`“ vorher abgebrochen hat. In dem Fall erhöhen Sie die Suchtiefe bzw. den Spin zur Verfügung gestellten Speicher. Einen mit zu geringer Suchtiefe / zu wenig Speicher gefundenen Fehlertrail, können Sie natürlich trotzdem benutzen!
- Modellierungen müssen die verlangten Eigenschaften sinnvoll realisieren, verständlich sein und keinen unvernünftig großen Zustandsraum bedürfen.

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Weitere Informationen finden Sie unter: https://www.informatik.tu-darmstadt.de/studium_fb20/im_studium/studienbuero/plagiarismus/index.de.jsp

Aufgabe 1: Petri-Netz (10 Punkte)

Kontext: Petri-Netze werden gerne zur Verifikation verteilter Programme eingesetzt. In dieser Aufgabe soll jedoch ein Petri-Netz mit PROMELA modelliert werden, und zwar *ohne* Nebenläufigkeit, mit nur einem Prozess.

Gegeben ist das Petri-Netz eines Keks-Automaten[1], bei dem Kekse aus dem „Keksspeicher“ durch 1€ Münzen aus der „Geldbörse“ gekauft werden können. Die Modellierung abstrahiert dabei sowohl eine 1€ Münzen als auch einen Keks durch einen Marker (ausgefüllter schwarzer Kreis).



Definition der Funktionsweise:

- Ein Petri-Netz hat zwei Arten von Knoten: Plätze (als Ellipsen dargestellt) und Transitionen (als Rechtecke dargestellt).
- Jeder Platz enthält eine nicht-negative Anzahl an Markern (dargestellt mit ausgefüllten Kreisen). Im Laufe der Ausführung kann sich diese Anzahl ändern.
- Eine Transition hat eingehende und ausgehende Kanten von und zu Plätzen. Eingehende Kanten beginnen an einem Platz (*eingehender Platz*) und enden in der Transition, ausgehende Kanten beginnen in der Transition und enden an einem Platz (*ausgehender Platz*).
- Wenn eine Transition ausgeführt wird, dann wird von jedem eingehenden Platz genau ein Marker entfernt und zu jedem ausgehenden Platz genau ein Marker hinzugefügt. Eine Transition kann nur ausgeführt werden, wenn die Anzahl der Marker an eingehenden Plätzen ausreicht.

  Beispiel: Die Transition Kaufaktion kann nur ausgeführt werden, wenn es jeweils mindestens einen Marker in den eingehenden Plätzen Einwurfschlitz und Keksspeicher gibt.

Zur Illustration der konkreten Funktionsweise gibt es von uns hier eine kurze Animation [2].

Eine vollständige Lösung muss folgende Datei im Verzeichnis `aufgabe1/` beinhalten: `petrinet.pml`

Aufgabe:

a) Modellieren Sie das oben gegebene Petri-Netz des Keksautomaten in PROMELA.

   Vorgabe: Array **short** places[5] soll für jeden der 5 Plätze (Geldbörse, Einwurfschlitz, Keksspeicher, Entnahmefach, Kasse) die Anzahl der Marker enthalten. Die Transitionen werden dann in einer Schleife nacheinander ausgeführt, dabei soll der Name der Transition Geld-Einwurf, -Rückgabe, usw. mit **printf** () ausgegeben werden.

b) Stellen Sie sicher, dass Ihre Modellierung keine Invalid-Endstates enthält (ändern Sie dafür ggf. Ihre Modellierung).

c) Spezifizieren und verifizieren Sie folgende *Invariante*[3] mithilfe von Assertions (und falls notwendig Ghostvariablen):

   Zu keiner Zeit, hat ein Platz eine negative Anzahl an Markern.

d) Spezifizieren und verifizieren Sie folgende Invariante mithilfe von Assertions (und falls notwendig Ghostvariablen):

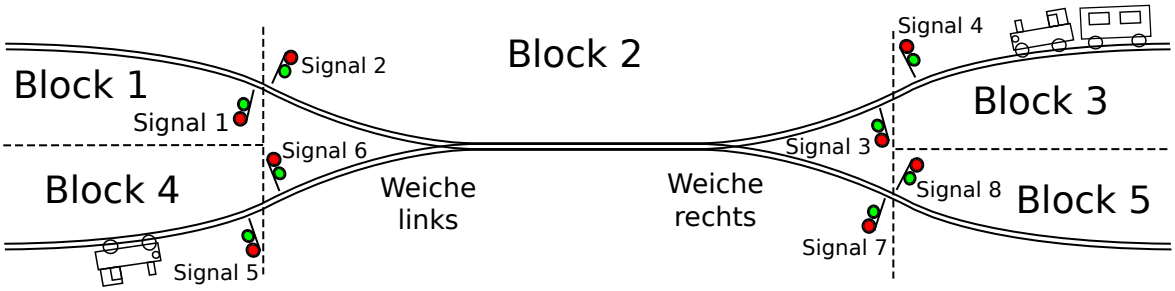
   Das Geld im gesamten System bleibt gleich, also die Summe der Marker in Geldbörse, Einwurfschlitz und interner Kasse.

Die fertige Modellierung inklusive der beiden Invarianten speichern Sie bitte als `petrinet.pml` ab.

[1] Modfiziert von: Wolfgang Reisig (7. Dezember 2010), Wikimedia, `https://commons.wikimedia.org/wiki/File:Petri2.png`
[2] `https://moodleload.hrz.tu-darmstadt.de/FB20_SE/FMiSE/Lab1/Kekse.mp4`
[3] Aussage, die in jedem Zustand des Systems gelten soll.

Aufgabe 2: Eingleisiger Streckenabschnitt (20 Punkte)

Kontext: Mit den neuesten Planungen für Stuttgart-21 wird die Digitalisierung von Zugsicherung und Zugsteuerung vorangebracht. Dass die Materie komplex ist, zeigen schon vermeintlich einfache Beispiele. Ziel dieser Aufgabe ist die Modellierung und Verifikation eines eingleisigen Streckenabschnitts. Der Modellierungsansatz ist dabei in der Datei `aufgabe2/single-track.pml` vorgegeben. Die zugehörige Grafik:



Es gibt zwei Weichen, eine linke, eine rechte und fünf sogannte Blöcke für die Zugsicherung, in der Grafik durch gestrichelte Linien geteilt. Die Zugsicherung gibt vor, dass jeder Block von maximal einem Zug belegt werden darf. In der Praxis könnte dies durch Signale zwischen den Blöcken gelöst werden, wie in der Grafik angedeutet. Die vorgegebene Datei enthält allerdings (noch) keine Signale.

Die beiden Weichen befinden sich beide in Block 2. Wenn die linke Weiche den Zustand 0 hat, dann ist Block 2 nach links mit Block 1 verbunden, bei Zustand 1 ist Block 2 nach links mit Block 4 verbunden. Wenn die rechte Weiche Zustand 0 hat, dann ist Block 2 nach rechts mit Block 3 verbunden, sonst mit Block 5. Die Weichenstellung betrifft dabei nur eine Richtung: Unabhängig von der Weichenstellung kann die linke Weiche stets in Richtung Links-nach-rechts und die rechte Weiche stets in Richtung Rechts-nach-links überfahren werden. Im vorgegebenen Modell sind beide Weichen im Zustand 0.

Eine vollständige Lösung muss folgende Dateien im Verzeichnis `aufgabe2/` beinhalten:

- `single-track_a.pml`
- `single-track_b.pml`
- `single-track_b.pml.trail`
- `single-track_b_error.txt`
- `single-track_c.pml`
- `single-track_d1.pml`
- `single-track_d1.pml.trail`
- `single-track_d1_error.txt`
- `single-track_d2.pml`

Prüfen Sie vor Ihrer Abgabe unbedingt nach, dass die Dateien vollständig sind und zueinander passen, also z.B. dass der Fehlertrail in `single-track_b.pml.trail` mit dem Modell in `single-track_b.pml` nachspielbar (replay) ist.

2a) Zugziel (3 Punkte)

1. Vervollständigen Sie das bereitgestellte Modell `single-track.pml`, sodass mindestens ein `train`-Prozess gestartet wird.
2. Verifizieren Sie in Ihrem vervollständigten Modell mittels **Assertions**, dass ein Zug mit Richtung Links-nach-rechts immer in Block 3 oder Block 5 und ein Zug mit Richtung Rechts-nach-links immer in Block 1 oder Block 4 endet.

Geben Sie die verifizierte Datei als `single-track_a.pml` ab.

2b) Blockbelegung (5 Punkte)

Nun wollen wir verifizieren, dass niemals zwei Züge gleichzeitig einen Block belegen. Denn sonst besteht die Gefahr eines Zugunglücks durch Zusammenstoß!

1. Starten Sie abermals mit dem von uns bereitgestellten Modell `single-track.pml` und erweitern Sie es so, dass je ein Zug aus den beiden Richtungen Links-nach-rechts und Rechts-nach-links gestartet wird.
2. Erweitern Sie das Modell zur Verifikation mittels **Assertions** (und falls notwendig Ghostvariablen), dass niemals zwei Züge gleichzeitig einen Block belegen. Geben Sie dieses Modell als `single-track_b.pml` ab.
3. Die Verifikation sollte fehlschlagen. Geben Sie den Fehlertrail als `single-track_b.pml.trail` ab. Erklären Sie, mit Bezug auf das im Fehlertrail ausgegebene Gegenbeispiel, warum die Verifikation fehlschlägt. Geben Sie Ihre Erklärung als reine Textdatei unter `single-track_b_error.txt` ab.

2c) Stellwerk I (5 Punkte)

Zwischen den Blöcken sollen nun Signale implementiert werden, um einen Zusammenstoß auch wirklich zu vermeiden.

1. Starten Sie wie in Aufgabe b) je einen Zug aus beiden Richtungen.
2. Erweitern Sie das Modell mit Signalen. Für jeden Übergang von und zu Block 2 gibt es je ein Signal (zum Beispiel Signal 1 für Block 1 zu Block 2 und Signal 2 für Block 2 zu Block 1). Wenn der nächste zu befahrende Block belegt ist, dann muss das Signal auf „halt“ stehen, sonst darf es auf „frei“ oder „halt“ stehen.
3. Modifizieren Sie das Modell so, dass Züge nur bei freiem Signal in den nächsten Block einfahren und die Signale dabei umgeschaltet werden.
4. Die Verifikation von Aufgabe b) sollte nun erfolgreich sein, wenn die Option „Invalid-Endstates (deadlocks)“ deaktiviert ist.

Geben Sie Ihr verifiziertes Modell als `single-track_c.pml` ab.

2d) Stellwerk II (7 Punkte)

Der Zusammenstoß ist vermieden, allerdings sollen die Züge trotzdem ankommen und sich nicht gegenseitig blockieren (Deadlock).

1. Ausgehend vom Modell `single-track_c.pml` aus Aufgabe c), erklären Sie wie und warum sich die Züge gegenseitig blockieren. Geben Sie hierfür ein ggf. angepasstes Modell `single-track_d1.pml`, einen Fehlertrail `single-track_d1.pml.trail` und Ihre Erklärung des Fehlers `single-track_d1_error.txt` ab.
2. Implementieren Sie einen Mechanismus zum Verstellen der Weichen, sodass sich die Züge nicht mehr gegenseitig blockieren. Eine Weiche in Block 2 darf nur verstellt werden, wenn Block 2 frei ist.

   Hinweis: Dafür müssen Sie ggf. auch den Quellcode für die Signale anpassen.
3. Verifizieren Sie mit Spin, dass es nun keinen Fehlertrail aufgrund von Invalid-Endstates mehr gibt. Geben Sie Ihr Modell als `single-track_d2.pml` ab.

Aufgabe 3: Netzwerkkommunikation (30 Punkte)

Kontext: Ein häufig auftretendes Problem bei Netzwerkprotokollen ist der Umgang mit verloren gegangenen Nachrichten. Da Übertragungen und Netzwerke nicht perfekt sind (anders als Kanäle in PROMELA), müssen Protokolle damit umgehen.

In dieser Aufgabe sollen Sie ein solches, fehlerresilientes Protokoll modellieren. In dem Protokoll gibt es zwei Prozesse, P und Q. Beide fungieren sowohl als Sender als auch Empfänger von Daten, die wir hier Frames nennen. Frames enthalten in unserer Modellierung eine Sequenznummer (seqNr), eine Bestätigungsnummer (ackNr) und das Paket (packet), das übertragen werden soll.

Die Idee des Protokolls ist, dass Prozess P sicher stellt, dass alle Pakete empfangen werden, indem P alle versendeten Pakete speichert, bis sie bestätigt wurden. Im Falle, dass eine Nachricht verloren geht, werden alle nicht bestätigten Nachrichten erneut gesendet.

Aus Effizienzgründen ist die Anzahl der zwischengespeicherten Nachrichten auf maximal $\mathsf{seq}_{\max} + 1$ Nachrichten begrenzt. Der Wert $\mathsf{seq}_{\max}$ entspricht der höchsten zu vergebenden Sequenznummer bevor wieder mit 0 begonnen wird.

Jeder Prozess hat intern einen Zähler für Sequenznummern und erhöht diesen schrittweise: $0, 1, ..., \mathsf{seq}_{\max}$. Nach $\mathsf{seq}_{\max}$ geht es wieder mit 0 weiter. D.h. alle Additionen von Sequenz- und Bestätigungsnummern im Folgenden müssen entsprechend angepasst gedacht werden.

Da P maximal $\mathsf{seq}_{\max} + 1$ Nachrichten zwischenspeichert, können nie zwei unbestätigte Nachrichten von P die gleiche Sequenznummer haben.

Um unnötigen Nachrichtenverkehr zu vermeiden, werden Frames nicht nur zur Datenübertragung genutzt, sondern auch zum Bestätigen von Nachrichten. Die Bestätigungsnummer ackNr in einer Nachricht von (bspw.) P drückt aus, dass P alle Nachrichten bis einschließlich ackNr empfangen hat. Abbildung 1 zeigt den Aufbau eines Frames.

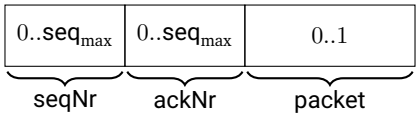


Abbildung 1: Aufbau des Frames „seqNr || ackNr || packet“ in unserem Protokoll

Abbildung 2 zeigt einen möglichen Kommunikationsablauf nach dem skizzierten Protokoll. P sendet vier Frames und bekommt erst dann eine Bestätigung von Q. Daraufhin erhöht P die Bestätigungsnummer um 1 (auf 0).

Eine vollständige Lösung muss folgende Dateien im Verzeichnis `aufgabe3/` beinhalten:
(i) `protocol1.pml`, (ii) `protocol2.pml` und (iii) `protocol3.pml`

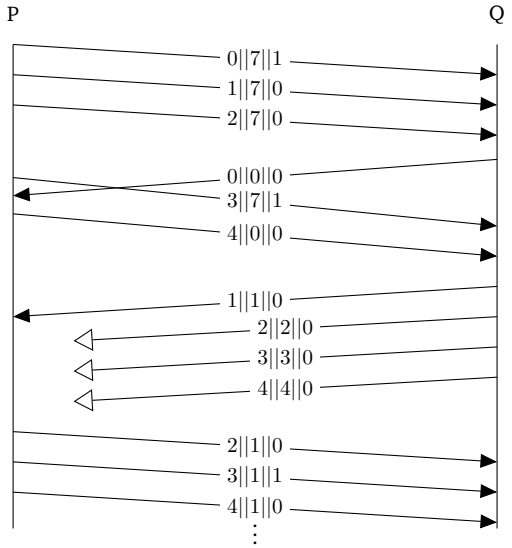


Abbildung 2: Beispielhafter Ablauf: P sendet vier Nachrichten bevor die erste Bestätigung ankommt. Eine Nachricht wie $0||7||1$ steht für einen Frame mit Sequenznummer 0, Bestätigungsnummer 7 und Paket 1. Die Bestätigungen der Nachrichten $2, 3$ und 4 gehen verloren, also sendet P diese Nachrichten erneut.

3a) Modellierung (25 Punkte)

Aufgabe: Vervollständigen Sie die in der Datei `aufgabe3/protocol.pml` vorgegebene PROMELA-Modellierung des im Folgenden beschriebenen Modells. Geben Sie Ihre Lösung unter `protocol1.pml` ab.

Protokollbeschreibung: Um das oben beschriebene Verhalten zu garantieren, muss Prozess P Wissen über den aktuellen Kommunikationsverlauf/-zustand vorhalten: Der Prozess speichert die Sequenznummer nextToSend, die der nächste Frame von P haben wird; die nächste Bestätigungsnummer nextAckExpected, die P erwartet zu empfangen; die nächste Sequenznummer frameExpected, die P erwartet, zu empfangen; und einen Buffer buffered für die versendeten, aber noch nicht bestätigten Pakete (packets). Der aktuelle Füllstand des Buffers (=Anzahl gebufferter Pakete) ist in der Variablen nBuffered festgehalten.

Der Prozess P verhält sich wie folgt:

(P1) P initialisiert zunächst die Variablen, die das für die Protokolldurchführung relevante Wissen über den Kommunikationsverlauf enthalten, mit dem Wert 0 bzw. den Buffer mit einem Array, das nur Nullen enthält.

(P2) Ist noch Platz im Buffer, dann wird

- (P2.1) ein zufälliges Paket packet generiert;
- (P2.2) das Paket packet gebuffert; und
- (P2.3) das Paket packet in einem Frame an Q gesendet.

(P3) Wenn eine Nachricht mit Frame $s||a||p$ im Netzwerk an P addressiert ist, dann:

- (P3.1) Falls die Sequenznummer s des empfangenen Frames erwartet wurde, wartet P ab jetzt auf $s + 1$. Ansonsten wird die Nachricht fallen gelassen.
- (P3.2) Alle Frames mit Sequenznummer kleiner-gleich a wurden bestätigt und werden aus dem Buffer entfernt.

(P4) Wenn es ein *Timeout* gibt, dann muss es einen Fehler in der Übertragung gegeben haben. Also senden wir die Pakete aller noch unbestätigten Frames erneut.

Hinweis: Die boolsche globale und nur lesbare Variable **timeout** hat den Wert **true**, falls alle laufenden Prozess blockiert sind, ansonsten hat sie den Wert **false**.

(P5) Gehe zu Schritt (P2).

Prozess Q modellieren wir als eine Vereinfachung von P. Er speichert nur seine nächste Sequenznummer nextToSend und die als nächstes erwartete Sequenznummer frameExpected. Der Prozess verhält sich wie folgt:

(Q1) Q initialisiert seine nächste Sequenznummer und die als nächstes erwartete Sequenznummer mit dem Wert 0.

(Q2) Q horcht am Netzwerk solange, bis eine an ihn addressierte Nachricht ankommt. Bei Empfang der Nachricht:

- (Q2.1) Aktualisiert Q den Wert der Variablen frameExpected auf die erwartete Sequenznummer, des nächsten zu empfangenden Paketes.
- (Q2.2) Sendet Q einen Frame mit Inhalt 0 an P und aktualisiert die Variable nextToSend mit der von Q als nächstes zu verwendenden Sequenznummer.
- (Q2.3) Geht Q zu Schritt (Q2).

Zur Kommunikation nutzen wir einen globalen Kanal network mit Kapazität MAX_SEQ, der zwei Werte überträgt: (i) die Prozess-Id des Prozesses, der die Nachricht empfangen soll und (ii) den zu übertragenden Frame.

Das Senden einer Nachricht über das Netzwerk sollte so ablaufen, dass ein Frame aus dem Paket gebaut wird und an die Prozess-Id des anderen Prozesses gesendet wird. Beachten Sie, dass die ackNr entsprechend gesetzt ist.

3b) Nicht-perfekte Kanäle (3 Punkte)

Kontext: Im Moment ist das Protokoll aus a) unnötig kompliziert, da PROMELA-Kanäle perfekt sind und keine Nachrichten verloren gehen. Der Buffer wird also nie gebraucht.

Aufgabe: Ändern Sie die Modellierung im Protokoll so, dass Nachrichten bei der Übertragung verloren gehen können. Geben Sie die Lösung unter `protocol2.pml` ab.

3c) Verifikation (2 Punkte)

Aufgabe: Spezifizieren und verifizieren Sie folgende Eigenschaft *ausschließlich unter Verwendung von Assertions (Zusicherungen) und Ghostvariablen (Geistervariablen)* für das Modell aus Aufgabe 3b). LTL-Formeln dürfen nicht verwendet werden.

> Für jeden Frame, der von Q versendet wird, gilt: Seine Sequenz- und Bestätigungsnummer sind identisch.

Geben Sie das Modell mit Ghostvariablen und Assertions als `protocol3.pml` ab.

Hinweise:

- Um die Verifikation zu vereinfachen, sollten Sie in dieser Teilaufgabe die maximale Sequenznummer auf 2 setzen. Sollten Sie ein select-Statement benutzen, schreiben Sie dieses (und **nur** dieses) Statement in einen **atomic**-Block. Das verbessert die Performanz.
- Prüfen Sie bei der Verifikation, dass Spin den gesamten Zustandsraum durchsucht hat und nicht wegen Speichermangel (`reached -DMEMLIM bound`) oder zu kleiner Suchtiefe (`max search depth too small`) vorher abgebrochen hat. In dem Fall kann es sein, dass Sie die Suchtiefe oder den Spin zur Verfügung gestellten Speicher erhöhen müssen; und/oder dass Sie prüfen sollten, ob ihre Modellierung einen unnötig großen Zustandsraum aufspannt.