»**LoLA**«

Anleitung zur Erstellung eines neuen Stores

Max Görner

1. November 2012

Betreuer:
Prof. Dr. rer. nat Karsten Wolf
Dr. ing. Niels Lohmann

Im Rahmen dieser Anleitung soll erläutert werden, wie für das Petrinetzanalyseprogramm »LoLA« ein neuer Store geschrieben werden kann. Dafür wird zunächst behandelt, was ein Store ist und wie ein Store strukturiert ist. Danach wird erklärt, wie ein neuer Store etabliert werden kann. Da fast alle Stores PluginStores sind, werden diese sowie deren Komponenten, NetStateEncoder und VectorStores, im Anschluss vorgestellt.

1 Allgemeine Stores

Um Petrinetze zu analysieren, müssen im Wesentlichen alle erreichbaren Netzzustände (sog. NetStates [1]) einmal besucht werden. Daher wird eine Datenstruktur benötigt, die es ermöglicht, für einen übergebenen NetState zu bestimmen, ob dieser vorher schon einmal übergeben wurde und sich diesen danach zu merken.

Stores bieten eine solche Datenstruktur über eine einheitliche Schnittstelle an. Dafür muss jeder Store von der Klasse »Store [2] « erben und folgende Funktionen implementieren:

get_number_of_calls Diese Funktion gibt die Anzahl der Aufrufe der Funktion »searchAndInsert« zurück.

get_number_of_markings Diese Funktion gibt die Anzahl der gespeicherten NetStates zurück. Diese wird neben anderen Zwecken zur Fehlersuche und zur Angabe der Programmgeschwindigkeit benötigt.

popState Diese Funktion entfernt eine Markierung aus dem Store und gibt sie zurück.

searchAndInsert Diese Funktion nimmt einen Zustand des Petrinetzes entgegen und speichert diesen gegebenenfalls. Außerdem gibt sie zurück, ob dieser Zustand schon früher übergeben wurde und reserviert gegebenenfalls Speicherplatz für Zusatzinformationen (sog. Payload). Diese Funktion ist eine der kritischsten Stellen des gesamten Programmes, da hier große Teile der Laufzeit verbraucht werden. Es ist daher erwünscht, dass diese Funktion threadsicher implementiert wird, um Nutzen aus Parallelisierungen ziehen zu können.

Weiterführende Details können den Kommentaren des Quellcodes oder der daraus abgeleiteten Doxygendokumentation entnommen werden.

2 Etablierung eines neuen Stores

Um einen neu geschriebenen Store verwenden zu können, muss für diesen ein neuer Kommandozeilenparameter eingeführt werden. Außerdem muss der Store in den Kompilierungsprozess und die Korrektheitstest integriert werden.

[1] Ein NetState ist eine Klasse, die sowohl die Anzahl der Markierungen auf jeder Stelle, als auch zusätzliche, zur Beschleunigung des Programmes dienende Informationen enthält. In Stores werden jedoch nur die Markierungsanzahlen gespeichert.

[2] Die Oberklasse aller Stores ist in »src/Stores/Store.h« definiert.

2.1 Kommandozeilenparameter einführen

Um einen neuen Kommandozeilenparameter einzuführen, muss diesem ein Store zugewiesen werden. Dazu muss in der Datei »src/Planning/Tasks.cc« in der Funktion »setStore()« ein neuer case-Block erstellt werden. Es bietet sich an, eine bestehende Zeile zu kopieren und den case-Schlüssel, z.B. »store_arg_psbbin«, sowie den zu verwendenden Store anzupassen. Hierbei sei angemerkt, dass »store_arg_« ein während der Kompilierung erzeugtes Präfix ist und nicht verändert werden darf. Der eigentliche Kommandozeilenparameter ist in diesem Fall »psbbin«.

Damit dieser Parameter auch erkannt und verwendet wird, muss er in der Datei »cmdline.ggo« in der Rubrik »Stores« als Value hinzugefügt werden.

Die bisherigen Kommandozeilenparameter wurden nach einem gewissen System erstellt. Die ersten Buchstaben bezeichnen hierbei die Art des Stores. Da es momentan nur PluginStores gibt, lauten die ersten Buchstaben immer »ps «. Der nachfolgende Buchstabe bezeichnet den verwendeten NetStateEncoder. Von jedem der 3 vorhandenen NetStateEncoder wird der erste Buchstabe verwendet. Die letzten Buchstaben wiederum bezeichnen den verwendeten VectorStore. Hier wurden einprägsame Kürzel für den jeweiligen Store gewählt. Es bietet sich an, den eigenen Kommandozeilenparameter auch nach diesem System zu wählen.

2.2 Kompilierungsprozess erweitern

Damit ein neu geschriebener Store kompiliert wird, müssen seine Quellcodedateien in die Liste »lola_SOURCES« in der Datei »src/Makefile.am« eingetragen werden.

2.3 Automatisierte Korrektheitstest

In »LoLA« wurden einige Korrektheitstest automatisiert.

Es gibt Tests, die prüfen, ob »LoLA« unter Verwendung der zu testenden Stores das erwartete Ergebnis berechnet. Es gibt ähnliche Tests, die dafür jedoch mehrere Threads verwenden und somit Fehler im Threading feststellen können. Abschließend gibt es noch Tests, die überprüfen, ob die Speicherverwaltung der Stores allen angeforderten Speicher wieder frei gibt.

Um einen Store in einen der oben genannten Tests einzubinden, muss in der Datei »tests/testsuit.at« in den entsprechenden Bereichen der entsprechende Befehl eingegeben werden.

Die Bereiche sind, entsprechend der obigen Reihenfolge, »AT_BANNER([Stores])« , »AT_BANNER([Parallel Stores])« und »AT_BANNER([Memory Management])«.

Die Syntax der Befehle ergibt sich aus den schon vorhandenen Befehlen und den Kommentaren in der genannten Datei. Es muss für jedes Netz, mit dem die entsprechende Eigenschaft überprüft werden soll, eine Anweisung erstellt werden.

Im Anschluss können mit »make check« die automatischen Korrektheitstest gestartet werden, in denen der neu integrierte Store auftauchen sollte.

3 PluginStores [3]

Es gibt sowohl für die Kodierung der NetStates, als auch die verwendete Datenstruktur mehrere Alternativen, die sich in verschiedenen Kontexten anbieten. So könnte bei sehr speicherplat-

[3] Die Oberklasse aller PluginStores ist in »src/Stores/PluginStore.h« definiert.

zintensiven Petrinetzen eine Komprimierung der NetStates erwünscht sein. Um nun die Vor- und Nachteile der Kodierungen und Datenstrukturen möglichst einfach kombinieren zu können, wurde das Konzept der PluginStores entworfen.

Ein »PluginStore« ist ein Store, der den oben umrissenen Bedingungen genügt. Allerdings verwendet er einen NetStateEncoder, um die NetStates wie gewünscht zu kodieren, und einen VectorStore, um Bitvektoren, das Resultat der Kodierung eines NetStates durch einen NetStateEncoder, abzuspeichern und wiederzufinden. Dadurch können prinzipiell alle Kodierungen mit allen Datenstrukturen kombiniert werden, wobei einzelne Komponennten in ihrem Funktionsumfang eingeschränkt sein können. Abbildung 1 verdeutlicht dies.

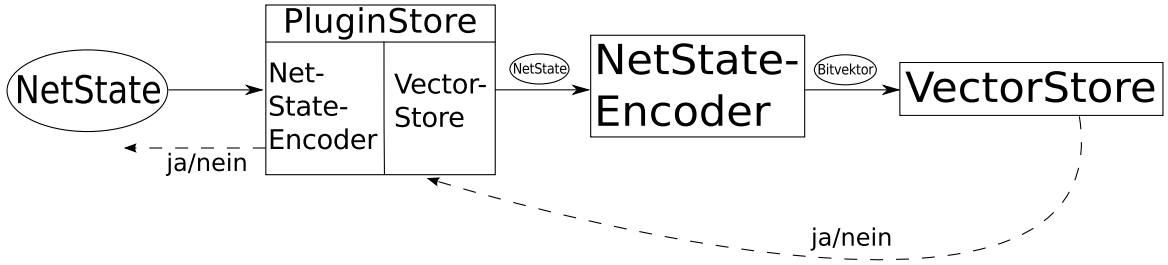

Abbildung 1: Diese Grafik zeigt den Ablauf der Verarbeitung eines Aufrufes der Funktion »searchAndInsert«.

Im folgenden soll daher erläutert werden, wie ein neuer NetStateEncoder und ein neuer VectorStore geschrieben werden können.

3.1 NetStateEncoder [4]

Ein NetStateEncoder (NSE) bildet einen NetState auf Bitvektor ab, damit dieser von einem VectorStore gespeichert werden kann. Die Länge der Bitvektoren muss hierbei keineswegs konstant sein.

Ziel der Abbildungen ist es, die unterschiedlichen Abwägungen verschiedener Einsatzgebiete zu Speicherverbrauch und Laufzeit zu berücksichtigen. Es gibt bisher folgende NSE:

FullCopyEncoder Dieser NSE kopiert alle Stellen eines NetStates.

CopyEncoder Dieser NSE kopiert nur die signifikanten Stellen eines NetStates und verwirft damit jene, die sich aus Stelleninvarianten ergeben. Dies spart etwas Speicherplatz, erschwert aber die Rekonstruktion des vollständigen NetStates.

BitEncoder Der BitEncoder arbeitet prinzipiell wie der CopyEncoder, verwendet aber weniger Bits für eine Stelle, falls eine Beschränkung der Anzahl der Marken bekannt ist. Bei Netzen mit niedriger Markenzahl ergeben sich dadurch sehr hohe Kompressionsgrade.

SimpleCompressedEncoder Dieser NSE kodiert die einzelnen Stellen mit logarithmischer Länge und nutzt damit aus, das kleine Zahlen besonders häufig vorkommen. Dies reduziert den Speicherverbrauch um einen Divisor bis zu 10, verlängert aber auch die Laufzeit erheblich.

[4] Die Oberklasse aller NetStateEncoder ist in »src/Stores/NetStateEncoder/NetStateEncoder.h« definiert.

Die genauen Geschwindigkeitseinbußen und Speicherplatzgewinne hängen von der konkreten Struktur des untersuchten Petrinetzes ab.

Ein NetStateEncoder muss die Funktion »**encodeNetState**« bereitstellen, welche einen übergebenen NetState auf einen Bitvektor abbildet und diesen zurück gibt.

Ein NetStateEncoder kann die Funktion »**decodeNetState**« bereitstellen, welche die Umkehrfunktion zur Abbildungsfunktion »encodeNetState« darstellt. Sie übernimmt damit einen Bitvektor und gibt einen NetState zurück.

Weiterführende Details können den Kommentaren des Quellcodes oder der daraus abgeleiteten Doxygendokumentation entnommen werden.

Damit ein neu geschriebener NetStateEncoder kompiliert wird, müssen seine Quellcodedateien in die Liste »lola_SOURCES« in der Datei »src/Makefile.am« eingetragen werden.

3.2 VectorStores [5]

Ein VectorStore ist kein Store im oben beschriebenen Sinne, da VectorStores nicht von der Klasse Store erben. Ansonsten ähnelt die Funktionsweise von VectorStores der von Stores.

Der wesentliche Unterschied besteht darin, dass VectorStores nicht NetStates sondern von NetStateEncodern als Bitvektoren kodierte NetStates speichern. Außerdem kümmern sie sich nicht um die Zählung von Aufrufen und Markierungen. Damit gibt es folgende Funktionen:

popVector Diese Funktion entfernt einen von NetStateEncodern als Bitvektoren kodierten NetState aus dem VectorStore und gibt ihn zurück.

searchAndInsert Diese Funktion nimmt einen Bitvektor entgegen und speichert diesen gegebenenfalls. Außerdem gibt sie zurück, ob dieser Bitvektor schon früher übergeben wurde und reserviert gegebenenfalls Speicherplatz für Zusatzinformationen (sog. Payload). Wie bei Stores ist diese Funktion die geschwindigkeitskritischste Funktion im gesamten Programm und sollte daher threadsicher sein.

Die Konzentration auf Bitvektoren ist wichtig, um Speicherplatz zu sparen. Die Funktion »searchAndInsert« nimmt dafür einen Zeiger und eine Länge in Bits entgegen. Nun können Datenstrukturen entwickelt und verwendet werden, die bitexakt arbeiten.

Ein leicht verständliches Beispiel ist eine Liste. Werden Listen als großer zusammenhängender Speicherbereich implementiert, in den die Elemente hintereinander geschrieben werden, ist es ohne weiteres möglich, Bitvektoren direkt hintereinander zu schreiben. Damit beträgt der größtmögliche Verschnitt 7 Bit und ist damit vernachlässigbar.

Folgende VectorStores existieren bisher:

BloomStore Dieser VectorStore basiert auf Hashfunktionen. Dadurch verbraucht er sehr wenig Speicher, kann aber falsche Antworten geben.

SuffixTreeStore: Dieser VectorStore kann Vektoren ohne Verschnitt speichern. Er ist sehr schnell und speichereffizient. Er verwendet Präfixbäume als Datenstruktur.

VSTLStore: Dieser VectorStore verdeutlicht das Prinzip von Stores. Sein didaktischer Nutzen ist größer als der für den produktiven Einsatz. Er verwendet inten STL-Sets, in denen STL-Vektoren gespeichert werden, als Datenstruktur.

[5] Die Oberklasse aller VectorStores ist in »src/Stores/VectorStores/VectorStore.h« definiert.

Weiterführende Details können den Kommentaren des Quellcodes oder der daraus abgeleiteten Doxygendokumentation entnommen werden.

Damit ein neu geschriebener VectorStore kompiliert wird, müssen seine Quellcodedateien in die Liste »lola_SOURCES« in der Datei »src/Makefile.am« eingetragen werden.

3.3 Verwendung

Wenn der gewünschte NetStateEncoder und VectorStore vorhanden sind, müssen für die Erstellung eines neuen PluginStores keine neuen Dateien angelegt werden. Es reicht, eine neue Klasse zu erstellen und einer entsprechenden Variablen zuzuweisen. Beispiele finden sich in Listing 1 und in der Datei »src/Planning/Tasks.cc« .

4 Templates und Payload

Diverse Algorithmen von »LoLA« erfordern es, NetStates nicht nur zu speichern, sondern zu diesen auch Zusatzinformationen, sogenannte Payloads, hinterlegen zu können. Aus diesem Grund verwenden alle Stores Templates. Diese ermöglichen es, dass jede Art Payload, von primitiven Datentypen bis hin zu ganzen Klassen, hinterlegt werden kann. Dieses Kapitel beschreibt die bisherigen Konventionen, die eingehalten werden sollten.

Die Verwendung von Templates beschränkt sich momentan darauf, die Signaturen der betroffenen Funktionen, sowie die Art gewisser Variablen und Rückgabetypen variabel zu halten. Bis auf eine Ausnahme, »Store<void>« verändern Templates momentan nicht die Arbeitsweise der Stores.

Damit im Falle des Verzichts auf Payload kein Speicherplatz verschwendet wird, wurde »void« als Schlüsselwort dafür definiert, dass kein Payload hinterlegt werden wird. Dementsprechend wird mit

Listing 1: Erstellung eines PluginStores

```
s = new PluginStore<void>(
    new SimpleCompressedEncoder(number_of_threads),
    new SuffixTreeStore<void>(), number_of_threads);
```

der Variablen s ein PluginStore, der intern einen SuffixTreeStore verwendet, aber keinerlei Speicherplatz für Payload verbraucht, zugewiesen. Es wird nicht einmal Speicherplatz für Variablen verbraucht, die nur für die Verwaltung des Payload interessant wären.

Durch die Verwendung von Templates ist es in den betroffenen Klassen nicht möglich, die übliche Aufteilung in einen Definitionsteil (.h-Datei) und einen Implementierungsteil (.cc-Datei), wobei die .h-Datei in der .cc-Datei inkludiert wird, beizubehalten. Eine ausführliche Erklärung dazu findet sich im Buch »C++ Templates: The Complete Guide« von Josuttis und Vandevoorde. Um trotzdem eine ähnliche Struktur beibehalten zu können, gibt es eine kurze und übersichtliche Datei mit Definitionen (.h-Datei), sowie einer Datei mit den Implementierungen (.inc-Datei), wobei am Ende der ersten die zweite inkludiert wird. Dies entspricht dem »Inclusion Model« aus dem genannten Buch.