»**LoLA**«

Anleitung zur Erstellung eines neuen Stores

Max Görner

29. Januar 2013

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 PluginStores [3]

Es gibt sowohl für die Kodierung der NetStates, als auch für die verwendete Datenstruktur mehrere Alternativen, die sich in verschiedenen Kontexten anbieten. So könnte bei sehr speicherplatzintensiven Petrinetzen eine Komprimierung der NetStates erwünscht und der damit verbundene Geschwindigkeitsverlust hinnehmbar sein, während in anderen Fällen schnellstmögliche Ausführungsgeschwindigkeit gewünscht ist. Um nun die Vor- und Nachteile der verschiedenen

[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.

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

Kodierungen und Datenstrukturen möglichst einfach kombinieren zu können, wurde das Konzept der PluginStores entworfen.

Ein »PluginStore« ist ein Store, der den in Kapitel 1 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. Einzelne Komponennten können jedoch in ihrem Funktionsumfang eingeschränkt sein. Abbildung 1 verdeutlicht das Prinzip der PluginStores.

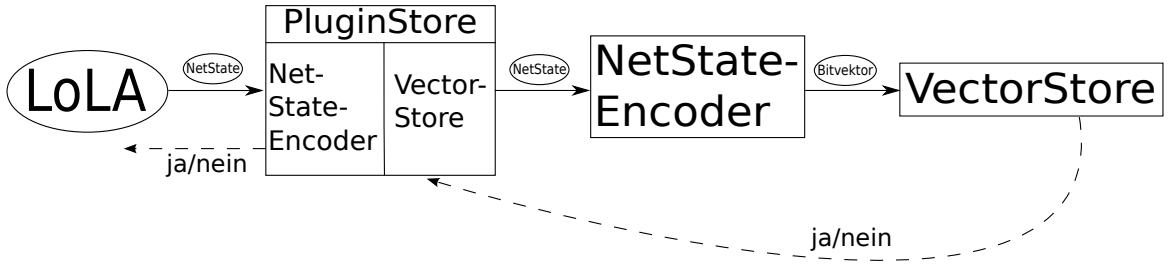

Abbildung 1: Diese Grafik zeigt den Ablauf der Verarbeitung eines Aufrufes der Funktion »searchAndInsert«.

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

2.1 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 kann Speicherplatz sparen, erschwert dann aber die Rekonstruktion des vollständigen NetStates.

BitEncoder Der BitEncoder arbeitet prinzipiell wie der CopyEncoder. Sollte für die Anzahl der Marken auf den Stellen des zu untersuchenden Petrinetzes eine obere Schranke s angegeben worden sein, verwendet es nur $\lceil \log_2(s) \rceil$ viele Bits zur Kodierung der Markenanzahl. Bei Netzen mit kleiner oberer Schranke ergeben sich dadurch sehr hohe Kompressionsgrade.

SimpleCompressedEncoder Dieser NSE kodiert die einzelnen Stellen mit logarithmischer Länge und nutzt damit aus, dass 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.

2.2 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 sparen zu können. 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 für eine solche Datenstruktur 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 vernachlässigbar.

Folgende VectorStores existieren bisher:

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

CompareStore Dieser VectorStore vergleicht die Ergebnisse zweier Stores. Damit können neue Stores einfach auf Korrektheit getestet werden. Auf diesen Store wird in Kapitel 4.3.1 näher eingegangen.

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

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

SuffixTreeStore: Dieser VectorStore kann Vektoren ohne relevanten 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 als Datenstruktur STL-Sets, in denen STL-Vektoren gespeichert werden.

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.

2.3 Verwendung von PluginStores

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.h« .

3 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 im Listing 1

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.

3.1 Bucketing

Ein weiterer, besonderer VectorStore ist der »HashingWrapperStore«[6]. Dieser verwaltet intern sehr viele VectorStores gleichen Typs und versucht die zu speichernden NetStates auf diesen möglichst gleichmäßig zu verteilen. In diesem Abschnitt werden zunächst die Vor- und Nachteile dieser Technik erläutert. Im Anschluss wird erläutert, wie Bucketing bei der Erstellung neuer Stores genutzt werden kann.

3.1.1 Vor- und Nachteile

Bei allen bisherigen Stores, mit Ausnahme des BloomStores, hängt die Zeit für einen Aufruf der Funktion »searchAndInsert« logarithmisch von der Anzahl der gespeicherten NetStates ab. Diese Anzahl wird oft sehr groß. Eine Beschleunigung[7] mit nur konstantem Speichermehrverbrauch kann daher erreicht werden, wenn die Markierungen durch geeignete Hashfunktionen sehr schnell auf viele verschiedene Stores aufgeteilt werden können, da die Anzahl der gespeicherten Netzzustände pro Store damit stark reduziert wird. Diese Aufteilung übernimmt der HashingWrapperStore.

Zusätzlich zur direkten Beschleunigung ist der HashingWrapperStore auch eine einfache Variante, mit beliebigen VectorStores eine Parallelisierung zu erreichen, da die einzelnen Stores voneinander unabhängig sind.

Allerdings wird bei der Verwendung des HashingWrapperStores mehr Arbeitsspeicher verbraucht. Schon die Erstellung und Vorhaltung sehr vieler Stores verbraucht etwas Speicherplatz. Dieser nur konstante Mehrverbrauch dürfte jedoch in den meisten Fällen vernachlässigbar sein. Viel gewichtiger ist, dass einige Stores, momentan der PrefixTreeStore und SuffixTreeStore, die Netzzustände speicherschonend abspeichern. Bei den beiden genannten Stores werden beispielsweise gleiche Anfänge in der Bitdarstellung zweier Vektoren nur genau ein Mal gespeichert. Diese Einsparung ist nicht mehr möglich, wenn Vektoren mit gleichen Anfangsstücken in unterschiedlichen Buckets landen.

3.1.2 Verwendung des Bucketing

Bucketing wird durch den Parameter »--bucketing« angestellt. Wie selbstgeschriebene Stores in »LoLA« integriert werden, wird im Kapitel 4 erläutert. Hier wird erläutert, wie die Erstellung von HashingWrapperStores funktioniert.

Ein HashingWrapperStore ist ein auf Templates basierender Store, dessen Konstruktor einen VectorStoreCreator[8] (VSC) entgegennimmnt. Ein VSC wird benötigt, da eine variable Anzahl Stores erstellt werden muss. Es genügt daher nicht, den gewünschten Store zu

[6] Der HashingWrapperStore ist in der Datei »src/Stores/VectorStores/HashingWrapperStore.h« definiert.

[7] Diese Beschleunigung ändert nicht die asymptotische Komplexität sondern beschleunigt um einen konstanten Faktor.

[8] Die Definition findet sich in der Datei des HashingWrapperStores.

übergeben; vielmehr wird eine beliebig oft aufrufbare Funktion benötigt, die den gewünschten Store übergibt. VSC übernehmen genau diese Aufgabe.

Da die Konstruktoren verschiedener Stores eine unterschiedliche Anzahl von Parametern erwarten, gibt es verschiedene VSC, namentlich den »NullaryVectorStoreCreator», den »UnaryVectorStoreCreator« sowie den »BinaryVectorStoreCreator«[9], die VectorStores mit keinem bis zwei Parametern erstellen. Eine Unterstützung weiterer Parameteranzahlen ist denkbar, aber bis jetzt nicht notwendig gewesen. Mit dem schon vorhandenen Quellcode sollten weitere VSC einfach implementiert werden können.

Ein konkretes Beispiel zur Erstellung eines PluginStores mit Bucketing ist in Listing 2 zu sehen.

Listing 2: Erstellung eines PluginStores mit Bucketing. Dem HashingWrapperStore wird ein UnaryVectorStoreCreator übergeben, welcher einen VSTLStore erstellt. Das Argument des VSC-Konstruktors wird an jeden erstellten VSTL-Store weitergereicht.

```
s = new PluginStore<T>(
        new BitEncoder(number_of_threads),
        new HashingWrapperStore<T>(
                new UnaryVectorStoreCreator
                        <T,VSTLStore<T>,index_t>
                        (number_of_threads)
                ),
        number_of_threads);
```

4 Etablierung eines neuen Stores

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

4.1 Kommandozeilenparameter einführen

Neue Kommandozeilenparameter werden auf einfache Weise in der Datei »cmdline.ggo« angegeben.

Soll ein Parameter für einen neuen Store erstellt werden, muss für diesen ein Parameter in der Rubrik »stores« hinzugefügt werden.

Soll ein Parameter für einen neuen NetStateEncoder erstellt werden, muss für diesen ein Parameter in der Rubrik »encoder« hinzugefügt werden.

Um mit den so definierten Kommandozeilenparametern die neuen NetStateEncoder aufrufen zu können, mussen in der Klasse »StoreCreator« in den Dateien »src/Planning/Tasks.h« und »src/Planning/Tasks.cc« die Funktion »createStore()« erweitert werden. Dies geschieht, indem in beiden Dateien der entsprechende switch-case-Block erweitert wird. Der anzugebende

[9] Alle definiert in der Datei des HashingWrapperStores.

case-Schlüssel besteht hierbei aus dem Präfix »encoder_arg_« und dem in der Datei »cmdline.ggo« eingetragenen Parameter.

Um neue Stores verwenden zu können, muss der entsprechende switch-case-Block in der Task.h erweitert werden. Der anzugebende case-Schlüssel besteht hierbei aus dem Präfix »store_arg_« und dem in der Datei »cmdline.ggo« eingetragenen Parameter. Sollte beim neuen Store die Verwendung von Hash-Buckets sinnvoll sein, muss dies innerhalb des case-Blocks unterschieden werden. Dies kann durch Abfrage, ob die Variable »args_info.bucketing_given« gesetzt ist, herausgefunden werden. Um dann einen Hash-Buckets verwendenden Store zu erstellen, muss dem PluginStore als VectorStore ein HashingWrapperStore übergeben werden.

Sollte der neu erstellte Store Payload nicht unterstützen können, muss statt der Task.h die Datei Task.cc angepasst werden. Hier muss die Spezialisierung der Funktion »createSpecializedStore« um den neugeschriebenen Store erweitert werden. Dies geschieht wiederum durch Einfügen eines neuen case-Schlüssels. Auf dies Weise wird sichergestellt, dass der neue Store nur aufgerufen werden kann, wenn Anwendungen, z.B. Erreichbarkeitsprüfung, keinen Payload erfordern.

4.2 Kompilierungsprozess erweitern

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

4.3 Korrektheitstest

Gerade bei der Entwicklung neuer Funktionen können Fehler gemacht oder entdeckt werden. Daher bietet »LoLA« eine Reihe von Maßnahmen, diese zu finden. In diesem Abschnitt werden die auf Stores bezogenen Qualitätssicherungsmaßnahmen behandelt.

4.3.1 Der CompareStore

Gerade bei der Entwicklung neuer Stores ist nicht immer ganz klar, ob diese Stores alle Netzzustände zuverlässig speichern und wiederfinden. Dies überprüft der CompareStore, indem er die Rückgaben zweier Stores vergleicht. Sind diese unterschiedlich, beendet sich das Programm.

Um einen zu überprüfenden Store anzugeben, muss dieser in der Task.h anstelle eines der momentan dort als Parameter des CompareStores eingetragenen Stores angegeben werden.

4.3.2 Automatisierte Korrektheitstests

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.