Integration des Model Checking Tools MARIA in die Petrinetz Entwicklungsumgebung RENEW

Friedrich Delgado Friedrichs

15. September 2006

Studienarbeit
Universität Hamburg,
Department Informatik,
Betreuer: Dr. Michael Köhler

Zusammenfassung

Die Überprüfung der Zuverlässigkeit von Informatik-Systemen ist seit der „Software-Krise“ der 1970er [Dijkstra, 1972] ein wichtiges Thema der Informatik. Neben dem Beweisen von Algorithmen und Code sind heute vor allem Testen bzw. Simulation und zunehmend auch Zustandsraum-Analyse (Model Checking) zur Systemverifikation gebräuchlich.

Diese Arbeit untersucht, ob und wie die beiden letzten Ansätze mit einer gemeinsamen Benutzungsoberfläche zusammengebracht werden können. Mit dem Petrinetz-Tool RENEW, das von sich aus bisher eher den Simulationsansatz verkörpert, sollen Petrinetze erstellt werden können, die vom Model Checking-Tool MARIA geprüft werden sollen.

Dabei sollte vor allem an Hand des erstellten Prototypen und dieses Textes eine bessere Einschätzung ermöglicht werden, welche Ansätze zur Integration von Analysemethoden in RENEW in Zukunft sinnvoll sind.

Das im Rahmen dieser Arbeit erstellte RENEW Plugin ermöglicht bereits die grafische Erstellung und den Import von nicht-modularen MARIA Netzen, die zusammen mit Anfragen an den Model Checker abgespeichert und von MARIA geprüft werden können. Weitere Ergebnisse dieser Arbeit sind zahlreiche Anregungen zur Weiterentwicklung des Plugins im Besonderen und zur Integration von Model Checking-Tools und -Verfahren in RENEW im Allgemeinen.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit stellt zunächst die beiden zu integrierenden Werkzeuge und ihre zugrunde liegenden Modelle vor:

Die Petrinetz Entwicklungsumgebung RENEW [Kummer u. a., 2006a] und der zugrunde liegende Netzformalismus wird kurz beschrieben. Mit RENEW können Referenznetze mit Anschriften in der Programmiersprache Java entwickelt und ausgeführt werden. Der Anwender kann eine Auswahl zwischen mehreren Petrinetz Formalismen unterschiedlicher Komplexität treffen und diese zum Teil mit der gesamten Ausdrucksmächtigkeit der Programmiersprache Java kombinieren. Das Verhalten des Systems kann anhand der Simulation studiert werden.

Dagegen bietet das Model Checking-Tool MARIA [Mäkelä, 2002] die Möglichkeit eine spezifische Ausprägung algebraischer Systemnetze [Kindler und Völzer, 1997] in einer syntaktisch an C angelehnten Sprache zu beschreiben. Der Zustandsraum des Systems kann mit Methoden des Model Checking untersucht werden. Die Möglichkeiten des Tools und des zugrunde liegenden Formalismus werden vorgestellt und mit denen von RENEW verglichen.

Dann werden mögliche Alternativen zur Integration von Model Checking in RENEW vorgestellt und die für die vorliegende Arbeit getroffene Auswahl begründet. Es wurde ein Kompromiss gewählt zwischen dem mächtigsten Netzformalismus und dem höchsten Implementierungsaufwand auf der einen Seite und der einfachsten Implementierung auf der anderen Seite.

In den dann folgenden Abschnitten wird die Implementation eines Prototyps konzipiert und der entwickelte Prototyp vorgestellt. Es wurde ein RENEW-Plugin entwickelt, das einen neuen Formalismus mit eigenen Netzanschriften und einem Anfragefeld implementiert. Die damit erstellten Netze werden aber nicht durch RENEW simuliert, sondern zusammen mit den Anfragen von MARIA ausgewertet.

Die Arbeit schließt mit einer Zusammenfassung der Arbeitsergebnisse und einem Ausblick auf mögliche Erweiterungen und Modifikationen des gewählten Ansatzes.

1.1 Typografische Konventionen

Quellcode, URLs, Kommandos sowie Prozessnamen werden in dieser Arbeit in `teletype` Schrifttypen gesetzt, die Namen von Programmpaketen, der LaTeX Konvention folgend, in KAPITÄLCHEN. Aus diesem Grund finden sich in diesem Text die Schreibweisen RENEW und MARIA, wenn die Programmpakete gemeint sind und `renew` und `maria`, wenn die konkreten Kommandos oder Prozesse gemeint sind.

2 RENEW: Die Referenznetz-Werkstatt

RENEW

Definition 1. RENEW *ist ein Werkzeug zur Entwicklung und Ausführung objektorientierter Petrinetze mit Netzinstanzen, synchronen Kanälen und nahtloser Integration der Programmiersprache Java zur vereinfachten Modellierung (nach [Kummer u. a., 2004]). Es wurde am Fachbereich Informatik der Universität Hamburg von Olaf Kummer, Frank Wienberg, Michael Duvigneau und einigen anderen entwickelt. Das Programm ist mit dem kompletten Java-Quellcode als Open-Source Software verfügbar und kann unter* `http://www.renew.de` *bezogen werden.*

In diesem Abschnitt wird zunächst informell über den von Olaf Kummer in [Kummer, 2002] vorgestellten Formalismus referiert, der später den von MARIA verwendeten algebraischen Systemnetzen gegenübergestellt werden wird.

Dann werden die für das Ziel dieser Arbeit wesentlichen Eigenschaften des Werkzeugs vorgestellt; zunächst die Plugin-Architektur, mit deren Hilfe neue Funktionalität in RENEW integriert werden kann. Im Rahmen dieser Arbeit wird ein Plugin erstellt werden, das einen neuen Netzformalismus für MARIA-Netze und je einen Menüpunkt zum Export von Netzen in die Petrinetz-Beschreibungssprache des MARIA Tools und zum Aufruf des Model-Checkers `maria` realisiert.

Da ein neuer Netzformalismus erstellt werden soll, wird dann auf die Implementation von Netzformalismen in RENEW eingegangen. Der hier zu erstellende Netzformalismus wird gegenüber den übrigen verfügbaren Formalismen zunächst unvollständig sein, da MARIA-Netze nicht simuliert sondern lediglich analysiert werden sollen. Aus diesem Grund muss auf den Simulator in dieser Arbeit *nicht* eingegangen werden.

2.1 Referenznetze

In [Kummer, 2002] werden Referenznetze formal spezifiziert. Die formale Konstruktion dort ist ausgesprochen umfassend und voraussetzungsvoll. Hier soll aber nur ein Vergleich der Formalismen in RENEW und MARIA erfolgen, mit dessen Hilfe entschieden werden kann, wie die beiden Programme zusammenarbeiten können.

An der informellen Darstellung der Formalismen zeigen sich schon die relevanten Unterschiede, so dass tiefer gehende theoretische Betrachtungen jedenfalls im Rahmen dieser Arbeit nicht erforderlich sind.

Obwohl Referenznetze in [Kummer, 2002] vollständig formal definiert sind, bis hin zur automatischen Freispeicherverwaltung von Netzexemplaren, erstreckt sich die formale Definition *nicht* auf die konkrete, auf Java basierende, Anschriftsprache. In diesem Text sollen mit dem Begriff „Referenznetze“ sowohl die in RENEW implementierten Referenznetz *mitsamt* Java Anschriften, als auch der reine Formalismus bezeichnet werden. Am häufigsten sind aber die Referenznetze mit Java-Anschriften gemeint. Zur Klärung wird wenn notwendig „Java-Referenznetze“, „formale Referenznetze“ oder eine ähnliche Formulierung verwendet werden. Andere in RENEW verfügbare, auch auf Referenznetzen basierende Formalismen, wie z. B. zeit-basierte Java Netze, SD-Netze oder Feature-Structure Netze bleiben in dieser Arbeit gänzlich unberücksichtigt.

Im Folgenden wird also der Begriff der Referenznetze informell wiedergegeben:

Referenznetze

Definition 2. ***Referenznetze****, wie sie in* RENEW *implementiert werden, sind gefärbte Petrinetze, die über synchrone Kanäle verfügen und deren Markierungen neben anderen (Java-) Datenobjekten auch Referenzen auf Netzexemplare enthalten können. Netzreferenzen dürfen hierbei nicht in initialen Markierungen enthalten sein, können aber beim Schalten von Transitionen erzeugt werden.*

synchroner Kanal
Uplink
Downlink

Definition 3. *Ein* ***synchroner Kanal*** *in diesem Sinne wird durch einen Uplink und einen Downlink beschrieben. Den* ***Uplink*** *spezifiziert eine Transitionsanschrift mit einer leeren Netzreferenz, einem Kanalnamen und optionalen Argumenten (z. B.* `:kanal(a,b)`*). Den* ***Downlink*** *beschreibt eine Transitionsanschrift mit einem Aus-*

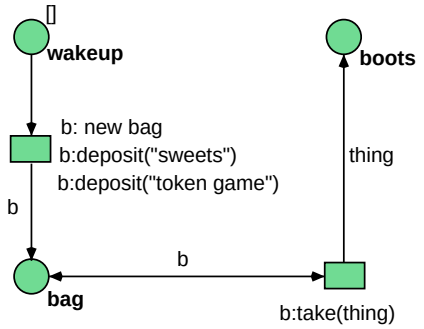

Abbildung 1: Das Netz „santa“

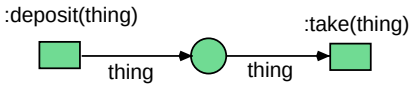

Abbildung 2: Das Netz „bag“

druck der zu einer Netzreferenz evaluieren muss, einem Kanalnamen der in dem referenzierten Netz als Uplink existieren muss und den entsprechenden aktualen Parametern (z. B. `andernetz: kanal(a,b)`*). (nach Kummer u. a. [2006b])*

Durch diesen Mechanismus können zum Einen Transitionen in Referenznetzen miteinander synchronisiert werden und zum Anderen wird (über die Kanalparameter) Datenübertragung zwischen Netzexemplaren ermöglicht.

In den Beispielnetzen[1] „santa“ und „bag“ (Abbildungen 1 und 2) werden diese beiden Mechanismen illustriert. In der Transition zwischen den Stellen „wakeup“ und „bag“ im Netz „santa“ wird zunächst eine Referenz auf ein neues Netzexemplar des Netzmusters „bag“ erzeugt[2] und dann zwei mal der Downlink `bag:deposit()` aufgerufen[3]. Dadurch schaltet die Transition mit dem Uplink `:deposit(thing)` in der erzeugten Netzinstanz `b` und das dem Kanalparameter übergebene Datenobjekt wird an den formalen Parameter `thing` gebunden und in der Ausgangsstelle der Transition im Netzexemplar `b` hinterlegt.

Neben der Erzeugung und Verwendung von Netzreferenzen als Marken und der Kommunikation über synchrone Kanäle können in RENEW viele Arten von Kanten und Inschriften verwendet werden. Reservekanten, Testkanten und flexible Kanten sowie virtuelle Stellen sind bereits aus anderen Petrinetzformalismen bekannt. Eine Besonderheit Java-basierter Referenznetze sind die „Action-Inschriften“ mit denen beim Schalten einer Transition beliebige Java-Ausdrücke ausgeführt werden können. (Siehe wiederum [Kummer u. a., 2006b] für die praktische Verwendung und [Kummer, 2002] für die theoretische Fundierung.)

Damit ist diese knappe und informelle Darstellung der Referenznetze abgeschlossen. Im Abschnitt 4 (S.15) wird zum Vergleich mit dem Netzformalismus des Tools MARIA hierauf zurück gekommen.

Es folgen nun zwei eher technisch orientierte Unterabschnitte, die im Hinblick auf die Implementation des Prototypen von Bedeutung sind.

[1] aus [Kummer u. a., 2006b]

[2] Terminologie nach [Kummer, 2002]. Netzexemplare sind konkrete Instanzen eines einzigen abstrakten Netzmusters, das u. U. in mehrfachen Exemplaren instantiiert sein kann.

[3] Die Reihenfolge in der die Transitionsanschriften evaluiert werden ergibt sich nebenbei bemerkt **nicht** aus der textuellen Reihenfolge. Siehe dazu [Kummer u. a., 2006b] und für eine detaillierte Beschreibung der Algorithmen [Kummer, 2002].

2.2 Ein Überblick über RENEWs Plugin-Architektur

RENEW basiert seit Version 2.0 auf einem Plugin-System. Die gesamte Anwendung wurde in mehrere Plugins zerlegt, die dynamisch geladen und entfernt werden können [Kummer u. a., 2006b].

Im Folgenden werden aus den RENEW Quellen und der Diplomarbeit von Jörn Schumacher [Schumacher, 2003] die Grundzüge des Plugin-Konzepts und seiner Verwendung erläutert.

Beim Start von `renew` wird als erstes die Methode `main` in der Klasse `de.renew.plugin.Loader` aufgerufen, welche dafür sorgt, dass wiederum die `main` Methode der Klasse `de.renew.plugin.PluginManager` gestartet wird, die die meisten anderen Komponenten der Software als Plugins lädt. Ein **Plugin** ist in diesem Sinne ein JAR-Archiv, das Java-Klassen und eine Konfigurationsdatei `plugin.cfg` enthält, in der die zur Einbindung des Plugins erforderlichen Informationen enthalten sind.

Plugin

Die Datei `plugin.cfg` liegt im Dateiformat der Klasse `java.util.Properties` vor und spezifiziert insbesondere die Hauptklasse des Plugins (`mainClass`), die nach dem Entwurfsmuster „Fassade“ das Initialisieren und Stoppen eines Plugins ermöglicht, den Namen des Plugins (`name`), den zur Verfügung gestellten Dienst (`provides`) und die vom Plugin benötigten Dienste (`requires`), außerdem einige weitere „Properties“, wie z. B. die Versionsnummer des Plugins.

Die Hauptklasse muss das Interface `IPlugin` implementieren, das der Plugin-Verwaltung (Klasse: `PluginManager`) die wichtigsten Methoden zur Verwaltung von Plugins zur Verfügung stellt. Wird die Klasse `PluginAdapter` erweitert, die ihrerseits `IPlugin` implementiert, müssen für ein einfaches Plugin lediglich noch die Methoden `init()`, zum Starten und `cleanup()` zum Stoppen des Plugins implementiert werden.

Die neuen Formalismen, sowie Menü- und Paletten-Elemente werden nun in der `init()`-Methode eingebunden und in der `cleanup()` Methode wieder entfernt. Der `PluginManager` sorgt dafür, dass diese Methoden beim Start bzw. beim Beenden des Plugins aufgerufen werden.

Das JAR-Archiv mit den Klassen des Plugins sowie der Konfigurationsdatei muss dann lediglich beim Programmstart in einem der Verzeichnisse liegen, die vom Loader nach Plugins durchsucht werden, damit das Plugin geladen und initialisiert wird. Das Anlegen und Kopieren des Archivs kann z. B. bequem mittels `ant` erfolgen.

2.3 Netzformalismen in RENEW

Aus Benutzersicht lassen sich Netzformalismen in RENEW mit Hilfe des Untermenüs „Formalisms“ im Menü „Simulation“ auswählen. Der gewählte **Formalismus** entscheidet, welche Anschriften für Netzelemente akzeptiert und welche Objekte vom Compiler erzeugt werden und darüber auch das Verhalten der Netze in der Simulation.

Formalismus

Über die Singleton-Klasse `FormalismPlugin` im Paket `de.renew.formalism` wird ein Formalismus eingebunden. Die Klassenmethode `getCurrent` liefert das aktuelle `FormalismPlugin` Objekt und an diesem wird mit einem `addCompilerFactory`-Aufruf eine Implementation von `ShadowCompilerFactory` angemeldet, welche wiederum mit der Methode `createCompiler` einen Compiler für ein Netz erzeugen kann. In der Regel haben diese konkreten Compiler das Präfix „`Single`“ und das Postfix „`NetCompiler`“ im Klassennamen. Sie sollten das Interface `ShadowCompiler` imple-

mentieren, welches Methoden zum Prüfen von Anschriften und zum Übersetzen des Netzes vorsieht.

Da die MARIA-Netze nicht simuliert, sondern nur exportiert und durch einen externen `maria` Prozess geprüft werden sollen, sind hier nur die Methoden zur Anschriftprüfung von Interesse.

Wie die Syntaxprüfung von Anschriften geschehen soll, schreibt das Interface nicht vor, es werden lediglich Methoden mit einem Rückgabewert der Klasse `String` deklariert. Diese werden von den verfügbaren Formalismen i. d. R. so verwendet, dass ein Parser erzeugt und aufgerufen und im Erfolgsfalle die Art der Inschrift als Text zurückgegeben wird. Bei Misserfolg wird eine vom Parser geworfene `ParseException` in eine `SyntaxException` umgewandelt, die von der RENEW-Oberfläche als Dialogbox dargestellt wird, so dass der Benutzer direktes Feedback über die Gültigkeit seiner Netzanschriften erhält.

Diese Art der Syntaxprüfung direkt beim Zeichnen eines Netzes soll auch vom zu entwickelnden `MariaNets` Plugin unterstützt werden. In den Erklärungen zu den Quelldateien `MariaParser.jj` (S. 28) und `SingleMariaNetCompiler.java` (S. 29) wird die konkrete Realisierung dieser Syntaxprüfung erläutert.

3 MARIA: modulare Erreichbarkeitsanalyse für algebraische Systemnetze

Das Model Checking-Tool MARIA von Marko Mäkelä [Mäkelä u. a., 2005] erlaubt die Spezifikation von algebraischen Systemnetzen mit einer an C und C++ angelehnten Sprache, in der auch komplexe Funktionen und Datenstrukturen spezifiziert werden können. Der Abschnitt 3.1 stellt die Sprache zur Netzbeschreibung vor.

Verschiedene Eigenschaften der Zustandsräume können in einer Anfragesprache abgefragt werden, die in Abschnitt 3.4 (S.13) kurz vorgestellt wird, nachdem in Abschnitt 3.3 zuvor einige Grundlagen zu Model Checking im Allgemeinen und bei MARIA erläutert wurden.

In diesem Abschnitt wird bewusst darauf verzichtet, eine genaue Anleitung zur Verwendung von MARIA zu geben, da dies bereits in [Mäkelä u. a., 2005] von den Autoren des Programms geschehen ist. Es wird lediglich ein Überblick über die Fähigkeiten des Programms gegeben. Für die Anwendung von MARIA und auch für die in dieser Arbeit entwickelte Anbindung an RENEW ist das Studium der Anleitungen und einiger zusätzlicher Literatur unverzichtbar.

3.1 Algebraische Systemnetze in MARIA

Algebraische Systemnetze werden im Artikel [Kindler und Völzer, 1997] als Erweiterung der von W. Reisig definierten algebraischen Petrinetze eingeführt. Unter anderem werden diese mit flexiblen Kanten versehen.

Wiederum wird die Definition algebraischer Systemnetze informal wiedergegeben, diesmal nicht, weil die formale Definition in [Kindler und Völzer, 1997, S.352f] zu umfangreich oder komplex wäre – sie ließe sich auf einer Seite bequem unterbringen, allerdings ohne die Schaltregeln – sondern weil die formale Darstellung für die Bewertung der konkreten Petrinetzsprache des Tools MARIA wenig helfen würde, da

die Flexibilität der allgemeinen algebraischen Systemnetze durch die konkreten Implementationsentscheidungen doch signifikant eingeschränkt wurden. (Die algebraische Definition der verwendeten Datentypen wird in [Mäkelä, 2001] gegeben.)

Gegenüber S/T-Netzen werden algebraische Systemnetze mit Netzanschriften versehen, die beliebige Operationen einer Algebra mit Bag-Signatur (BSIG-Algebra) erlauben, d. h. nach [Kindler und Völzer, 1997, S.352f]:

- die Stellen werden als Variablen aufgefasst, die Bags (positive Multimengen) von Sorten einer beliebigen SIG-Algebra als Werte annehmen können. Initiale Markierungen jeder Stelle können Terme der BSIG-Algebra sein, die keine freien Variablen enthalten,
- die Transitionen können als Anschriften Terme der BSIG-Algebra erhalten, die zu Werten der Sorte `bool` ausgewertet werden können
- und die Kanten können schließlich mit beliebigen Termen der BSIG-Algebra beschriftet werden.

Damit lassen sich beliebige Rechenoperationen mit algebraischen Systemnetzen durchführen, insbesondere wäre es auf dieser hohen Stufe der Allgemeinheit vermutlich noch möglich, Netzreferenzen und synchrone Kanäle mit Hilfe einer konkreten Algebra oder einer Erweiterung des Formalismus einzuführen, wenn auch die Spezifikation selbst solche Erweiterungen nicht vorsieht oder unterstützt. Zu prüfen, ob eine solche Abbildung von Referenznetzen auf algebraische Systemnetze tatsächlich machbar ist, und wie sie konkret umzusetzen wäre übersteigt allerdings den Rahmen dieser Arbeit bei Weitem. Im Vorhinein wäre auch zu klären, wozu eine solche Abbildung überhaupt dienen sollte. Im konkreten Fall haben wir es mit deutlich weniger abstrakten Klassen von BSIG-Algebren zu tun, wie im nächsten Unterabschnitt dargestellt wird.

3.2 Spezifikation von Petrinetzen in MARIA

Die Sorten der Algebra in den algebraischen Systemnetzen nach [Kindler und Völzer, 1997] finden sich bei MARIA in den vordefinierten und benutzerdefinierten Datentypen wieder und die Operatoren der Algebra in den vordefinierten Operatoren und den benutzerdefinierten Funktionen. (Referiert nach [Mäkelä u. a., 2005].)

In MARIA können mit Hilfe des Schlüsselworts `typedef` benutzerdefinierte Typen zusammengesetzt werden aus zwei Ganzzahltypen und einem Zeichentyp, welche genau den Typen `int`, `unsigned [int]` und `unsigned char` der Sprache C entsprechen, einem booleschen Typen und Aufzählungen. Die einfachen Typen können auch direkt benutzt werden. Eine Besonderheit der zusammengesetzten Datentypen besteht darin, dass sie in beliebiger Verschachtelung kombiniert werden können. Es finden sich die aus C entlehnten `struct` und `union`, weiterhin Arrays mit beliebigen einfachen Typen als Indextyp und „Queues“ (Warteschlangen) und „Stacks“ (Stapel), jeweils mit fester Kapazität. Der Wertebereich von einfachen Typen kann weiterhin durch „Constraints“ (Einschränkungen) verkleinert werden.

Es wurde bei allen Datentypen darauf Wert gelegt, dass sie einen beschränkten Wertebereich haben, eine (unabhängig von der Implementation eindeutig zu bestimmende) Totalordnung aufweisen und sich platzsparend abspeichern lassen [Mäkelä,

2001, S.25]. Diese Entwurfsentscheidungen waren insbesondere notwendig, weil MARIA den gesamten Zustandsraum (mit Einschränkungen auf Grund von Optimierungen) explizit entfaltet, was bei Datentypen mit unbeschränkter Größe und Kapazität zu beträchtlichen Implementationsproblemen geführt hätte.

Auf Zeiger und Referenzen wurde bewusst verzichtet, weil sie *alle* oben genannten Anforderungen und *zusätzlich* das Lokalitätsprinzip in Petrinetzen verletzen, was ebenfalls Probleme bei den verwendeten Verifikationstechniken bereitet. (Eine modulare Zerlegung des Zustandsraums anhand von Subnetzen oder Zusammenhangskomponenten kann z. B. an nicht-lokalen Effekten scheitern.) Stattdessen gibt es einen speziellen Typ `ID`. Dieser „Identifier"-Typ, eine Art Pointer der auf einen Zieldatentypen (und damit auf einen endlichen Wertebereich) beschränkt ist, ist der Sprache SDL (CCITT Specification and Description Language) entlehnt, einer Sprache die nach [Mäkelä, 2001] vornehmlich von der Telekommunikationsindustrie zur Spezifikation von Protokollen verwendet wird.

Eine große Zahl von Operatoren kann zur Bildung von Ausdrücken und Funktionen verwendet werden, wobei als einzige Kontrollstruktur lediglich der erweiterte `?:` Operator, aber keinerlei Schleifen zur Verfügung stehen (der Kontrollfluss einer Berechnung kann aber natürlich z. B. über Transitionen mit entsprechenden `gate`-Ausdrücken festgelegt werden[4]). Die Operatoren selbst sind zum großen Teil der Sprache C entlehnt und werden durch einige zum Teil recht mächtige Operationen auf Multimengen ergänzt, wie z. B. den Operator `map`, sowie prädikatenlogische Quantoren und einige andere. Nicht zuletzt sind in der Abfragesprache auch LTL-Ausdrücke möglich (siehe Abschnitt 3.3).

Die Petrinetzsprache von MARIA wird durch Ausdrücke zur Spezifikation von Stellen und Transitionen vervollständigt, sowie durch spezielle Anweisungen an den Model-Checker. Zu letzteren gehören „enabledness sets" und „fairness sets", die sowohl global als auch in Transitionsdefinitionen (neben den bereits im Formalismus der algebraischen Systemnetze vorgegebenen „Guards" [Kindler und Völzer, 1997, S.353]) angegeben werden können. Zu den Fairness-Bedingungen in MARIA-Netzen soll in Abschnitt 3.4 (S.13) etwas mehr gesagt werden.

Ebenfalls zu den Anweisungen an den Model-Checker gehören die speziellen Ausdrücke „`reject`", mit dem ein unerwünschter Zustand spezifiziert werden kann, sowie „`deadlock`", mit dem eine Warnung oder einen Fehler generiert werden kann, wenn ein Zustand ohne Folgezustände gefunden wird.

3.3 Model Checking

In diesem Unterabschnitt wird zunächst das Verfahren der Zustandsraum-Analyse allgemein und, im Hinblick auf MARIA, auch die „Linear Time Logic" (LTL) beschrieben, damit im nächsten Unterabschnitt auf die Praxis der Zustandsraum-Analyse mit MARIA eingegangen werden kann.

[4] In den algebraischen Systemnetzen nach [Kindler und Völzer, 1997] heißen die Bedingungen „guards", in MARIA hingegen `gate`.

3.3.1 Model Checking allgemein

In der Einführung zu Clarke und Schlingloff [2001] wird die folgende konzise Charakterisierung für Model Checking (Zustandsraum-Analyse) gegeben:

Model Checking

Definition 4. ***Model Checking*** *ist ein automatisches Verfahren, um Korrektheitseigenschaften sicherheitskritischer reaktiver Systeme zu verifizieren.*

[...]

reaktives System

Ein ***reaktives System*** *[...] besteht aus mehreren Komponenten, die dafür bestimmt sind, miteinander und mit der Umgebung des Systems zu interagieren. Im Unterschied zu funktionalen (oder transformationalen) Systemen, in denen die Semantik als Funktion von Eingabewerten nach Ausgabewerten gegeben ist, wird ein reaktives System durch seine temporalen Eigenschaften spezifiziert. Eine* ***(temporale) Eigenschaft*** *ist eine Menge von erwünschten Verhaltensweisen in der Zeit; das System erfüllt die Eigenschaft, wenn jede Ausführung des Systems zu der Menge gehört. Vom logischen Standpunkt aus gesehen wird das System durch eine semantische (Kripke-) Struktur beschrieben und die Eigenschaft durch eine logische Formel.*

(temporale) Eigenschaft

Nun sei noch die Definition einer Kripke-Struktur aus Valk [2004] zitiert:

Kripke-Struktur

Definition 5. *Eine* ***Kripke-Struktur*** *ist ein Tupel* $M := (S, S_0, R, L)$*, für das gilt:*

1. *S endliche Zustandsmenge,*
2. *$S_0 \subseteq S$ Menge von Anfangszuständen,*
3. *$R \subseteq S \times S$ links totale (Transitions-) Relation,*
4. *$L : S \to 2^{AP}$ Abbildung, die jedem Zustand s eine Menge $L(s) \subseteq AP$ von aussagenlogischen atomaren Formeln zuordnet (die in diesem Zustand gelten).*
5. *Ein Pfad oder eine Rechnung aus $s \in S$ ist eine Folge $\pi = s_0, s_1, s_2, \ldots$ mit $s_0 = s$ und $\forall i \geq 0 : R(s_i, s_{i+1})$).*

Kripke-Frame

Das Paar (S, R) wird auch als ***Kripke-Frame*** *bezeichnet.*

Alle möglichen Zustände eines Systems (wie z. B. eines Programms oder auch Petrinetzes) werden nun ausgefaltet. Die Ausfaltung eines Petrinetzes z. B. wäre natürlicherweise der Erreichbarkeitsgraph, wobei die Ausfaltung als Kripke Struktur dargestellt werden kann. (Die Knotenmenge des Erreichbarkeitsgraphen (also alle erreichbaren Markierungen) wäre dann die Zustandsmenge, die Relation R beschreibt die Kanten des Graphen und die Abbildung L wird Formeln ergeben, die Markierungen von Stellen oder Variablenbelegungen spezifizieren.)

Diese Zustandsmenge wird nun auf die Gültigkeit der (üblicherweise durch temporallogische Formeln) spezifizierten Eigenschaften hin untersucht.

Dabei entsteht typischerweise das Problem, dass die Zustandsmenge sehr groß werden kann (**Zustandsraumexplosion**), so dass der verfügbare Speicher nicht mehr

Zustandsraumexplosion

ausreicht oder die benötigte Rechenzeit nicht mehr akzeptabel ist. Damit auch komplexe Systeme mit vielen Zuständen analysiert werden können, bedient man sich zum Einen symbolischer Verfahren wie etwa „Binary Decision Diagrams“ und zum Anderen Optimierungsverfahren die sich auf die effiziente Speicherung von Zuständen, Ausnutzen von Symmetrien, strengen Zusammenhangskomponenten und anderen Eigenschaften des Systems beziehen.

3.3.2 Model Checking in MARIA

In MARIA wurde auf die Verwendung symbolischer Techniken verzichtet. Statt dessen wird der Zustandsgraph explizit entfaltet. Es werden aber einige Optimierungen eingesetzt. Z. B. werden strenge Zusammenhangskomponenten bei der Analyse ausgenutzt und statische und dynamische Heuristiken angewandt. Die genauen Techniken sind für diese Arbeit nicht weiter von Interesse und können in [Mäkelä, 2001] und [Mäkelä, 2003] nachgelesen werden.

Für die Spezifikation gewünschter oder unerwünschter Eigenschaften stehen neben den algebraischen Operationen der Spezifikationssprache (u. a. Boolesche Ausdrücke mit Existenz- und Allquantor, Operationen auf Multimengen, Arrays und Structs) auch Sicherheits- und Fairness-Bedingungen sowie LTL-Ausdrücke zur Verfügung, die im Folgenden genauer beschrieben werden.

Sicherheits Constraints

Sicherheits Constraints können mit Hilfe der Schlüsselwörter `deadlock` und `reject` spezifiziert werden.

Der `deadlock` Ausdruck wird in einem Zustand ohne Nachfolger ausgewertet, wenn er wahr wird, wird ein Fehler gemeldet. Der spezielle Ausdruck `fatal` kann überall benutzt werden, um die Analyse abzubrechen, so dass `deadlock fatal` die Analyse im Falle eines Deadlocks abbricht.

Wann immer der auf `reject` folgende Ausdruck wahr wird, wird ein Fehler gemeldet, auch hier kann mit `fatal` die Analyse abgebrochen werden.

Listing 1: Das Netz `dining.pn`

```
1  #!/usr/local/bin/maria
2  typedef unsigned (1..5) philosopher;
3  typedef struct {
4    philosopher p,
5    enum { thinking, hungry, eating } s
6  } status;
7  place fork (0..#philosopher) philosopher: philosopher p: p;
8  place state (#philosopher) status: philosopher p: { p, thinking };
9  trans left
10 in { place state: { p, thinking }; place fork: p; }
11 out { place state: { p, hungry }; };
12 trans right
13 in { place state: { p, hungry }; place fork: +p; }
14 out { place state: { p, eating }; };
15 trans finish
16 in { place state: { p, eating }; }
17 out { place state: { p, thinking }; place fork: p, +p; };
18
19 deadlock true;
```

Gegeben das Beispielnetz `dining.pn` aus dem MARIA-Handbuch [Mäkelä u. a., 2005, S.68] (Listing 1), wird der folgende Ausdruck (zu `deadlock fatal` äquivalent) die Analyse abbrechen, sobald die Stelle `fork` leer wird:

```
reject place fork equals empty && fatal
```

Fairness Bedingungen

Fairness Bedingungen können sowohl an Transitionen als auch für ein ganzes Netz mit den Schlüsselwörtern `weakly_fair`, `strongly_fair` und `enabled` notiert werden.

Wenn die Bedingungen direkt an einer Transition gegeben werden, dienen die booleschen Ausdrücke zur Identifikation der jeweiligen Instanz der Transition. Eine **Transitionsinstanz** ist dabei eine Transition mit dazugehöriger Variablenbindung. Im Beispielnetz aus Listing 1 auf der vorhergehenden Seite könnte beispielsweise zur Transition `left` die Mengenspezifikation `strongly_fair philosopher pl: pl==p` geschrieben werden, die alle Transitionsinstanzen mit den Bindungen `1==p`, `2==p`, `3==p`, `4==p` und `5==p` als eine Fairness-Menge behandelt. Der Model-Checker sorgt dann dafür, dass nur Zustandsübergänge gebildet werden, bei denen gewährleistet ist, dass jede Transitionsinstanz aus der Fairness-Menge, wenn sie unendlich oft aktiviert wird auch unendlich oft schaltet[5]. (Im Falle von `weakly_fair` muss die Transitionsinstanz *ständig* aktiviert sein, um unendlich oft zu schalten.) (Nach [Latvala, 2001], dort wird dieses Verhalten durch so genannte Faire Kripke Strukturen (FKS) formal beschrieben.)

Transitionsinstanz

Ein `enabled` Ausdruck definiert eine Menge von Transitionsinstanzen, von denen mindestens eine aktiviert werden muss. Wenn keine von den Instanzen in der Menge aktiviert werden kann, wird die Menge am Ende der Analyse ausgegeben.

Alle Fairness Bedingungen können statt an einer Transition auch als Netzanschriften gegeben werden, in dem Falle muss vorher mit `trans t:` spezifiziert werden, zu welcher Transition Instanzen spezifiziert werden sollen.

Außerdem können Fairness Bedingungen auch direkt in LTL spezifiziert werden, allerdings ist es effizienter, die Schaltregeln direkt im Modell zu spezifizieren, weil dadurch der Zustandsraum verkleinert wird, außerdem werden die LTL Formeln sonst zu komplex, was das Verfahren sehr ineffizient macht, da Model-Checking PSPACE-vollständig in Abhängigkeit von der Länge der Formel ist (nach Latvala [2001]).

LTL

LTL (Linear Time Logic) ist eine echte Teilmenge der CTL* (Erweiterung von CTL (Computation Tree Logic)), bei der alle Formeln als quantifiziert über alle Pfade betrachtet werden. Alle folgenden (d. h. vom aktuellen Zustand aus über die Transitionsrelation erreichbaren) möglichen Zustände der Kripke-Struktur werden sozusagen als ein linearer Zeitstrahl betrachtet. In MARIA lassen sich LTL-Ausdrücke mit den Operatoren

<> (*eventually*, in irgendeinem späteren Zustand),

[] (*henceforth*, in allen folgenden Zuständen),

() (*in the next state*, im nächsten Zustand),

until („bis“, d. h. `p until q`, in einem folgenden Zustand gilt `q` und in allen Zuständen davor gilt `p`) und

[5] Bei dem Beispielnetz nützt diese Spezifikation natürlich wenig, da es selbst bei einer strengen Fairness Bedingung nicht fair schaltet. Im Unterabschnitt 3.4 wird ein Beispiel gegenübergestellt, bei dem dies hilft.

release („freigeben“, d. h. `p release q`, es gilt immer `q` oder bis einschließlich des ersten Zustandes in dem `q` gilt, gilt `p`)

und den übrigen Ausdrücken der MARIA Sprache bilden.

3.4 Untersuchen von Zustandsräumen mit `maria` und `lbt`

In MARIA kann der Zustandsraum mit verschiedenen Algorithmen ausgefaltet und anschließend auch interaktiv erforscht werden, effizienter ist jedoch die Spezifikation von Anforderungen in LTL, die in dem nun folgenden praktischen Exkurs vorgeführt wird.

Zu diesem Zweck wird ein externes, separat zu installierendes Programm verwendet, welches LTL Ausdrücke in Büchi-Automaten übersetzt, die wiederum von MARIA ausgewertet werden. Das Programm LBT (erhältlich unter http://www.tcs.hut.fi/Software/maria/tools/lbt/) kann dafür verwendet werden und muss beim Start von MARIA mit der Option `-p` oder während der Laufzeit mit dem Befehl `translator` angegeben werden.

Zum Netz aus Listing 1 (S.11) lässt sich nun abfragen, ob in *allen* Pfaden mindestens ein Philosoph etwas zu essen bekommt:

```
<>(cardinality subset st {place state} (st.s==2)) > 0
```

Wenn der Befehl `lbt` nicht gefunden werden kann, wird ein Fehler gemeldet, andernfalls wird `property holds` gemeldet, wenn die Eigenschaft zutrifft, wenn nicht (wie in diesem Fall, auch mit Fairness-Bedingung) gibt MARIA ein Gegenbeispiel in Form eines Pfades aus.

Die Existenz eines Gegenbeispiels zeigt an, dass die Bedingung nicht in *allen* möglichen Pfaden gilt. Wie bereits gesagt, werden LTL Formeln immer für alle Pfade geprüft. Obwohl es also keine Möglichkeit gibt, die Existenz eines Zustands mit LTL zu prüfen, lässt sich in MARIAs Abfragesprache ein Pfad zu einem bestimmten Zustand mit dem Befehl `path` finden, in unserem Beispiel gibt

```
path (cardinality subset st {place state} (st.s==2)) > 0
```

den Pfad zu einem Zustand aus, in dem mindestens ein Philosoph isst und

```
path (cardinality subset st {place state} (st.s==2)) > 1
```

gibt aus, dass es keinen Pfad zu einem Zustand gibt, in dem mehr als ein Philosoph isst.

Mit einem vorangestellten `visual` können die Anfrageergebnisse bei vielen Befehlen auch im interaktiven grafischen Browser dargestellt werden, sofern die Programme `lefty` [6] und `maria-vis` im Suchpfad für Programme liegen und das Script `dotty.lefty` zur Verfügung steht. Allerdings hat die grafische Ausgabe das Problem, dass Pfade, die einen oder mehrere Zustände mehrfach durchlaufen in uneindeutiger Weise dargestellt werden.

Nachdem der Model Checker gemeldet hat, dass das Modell unerwünschte Eigenschaften hat, lässt sich das Modell anpassen und anschließend mit den gleichen Anfragen überprüfen. Darüber hinaus nehme ich noch einige Änderungen vor, die die Überprüfung etwas komfortabler machen sollen.

Das modifizierte Netz ist in Listing 2 auf der nächsten Seite zu sehen.

[6] aus dem GRAPHVIZ Paket (http://www.graphviz.org/)

Listing 2: Das Netz `dining-fair+safe.pn`

```
1  typedef unsigned (1..5) philosopher;
2  typedef enum { thinking, hungry, eating } state;
3
4  typedef struct {
5    state s,
6    philosopher p
7  } status;
8
9  place fork (0..#philosopher) philosopher: philosopher p: p;
10 place state (#philosopher) status: philosopher p: { thinking, p };
11 place plates (0..(#philosopher - 1)) unsigned: (#philosopher-1)#1;
12
13 trans left
14 in { place state: { thinking, p }; place fork: p; place plates: 1;}
15 out { place state: { hungry, p }; }
16 strongly_fair philosopher ph: ph==p;
17
18 trans right
19 in { place state: { hungry, p }; place fork: +p; }
20 out { place state: { eating, p };};
21
22 trans finish
23 in { place state: { eating, p }; }
24 out { place state: { thinking, p }; place fork: p, +p; place plates: 1;};
25
26 bool isinstate(philosopher p, state s)
27   (cardinality subset m {place state}
28         (m.s == s && m.p == p))
29    == 1;
30
31 bool iseating(philosopher p)
32   isinstate(p, eating);
33
34 bool isthinking(philosopher p)
35   isinstate(p, thinking);
36
37 bool ishungry(philosopher p)
38   isinstate(p, hungry);
39
40 deadlock true;
```

Zunächst ist die Abfrage `(st.s == 2)` nicht besonders klar. Es wird ein Wert einer *enum* Variable getestet, aber weil der entsprechende *enum* Typ innerhalb einer Typdefinition spezifiziert wird, kann nicht auf die Namen der Werte zugegriffen werden. Darum wird in Zeile 2 ein benannter *enum* Typ definiert und die folgende Definition der Struktur `status` wird entsprechend modifiziert.

Außerdem sind die obigen Anfragen viel Tipparbeit, MARIA kann zwar auch Anfragen an der Kommandozeile mit dem Schalter `-e` (bzw. `-execute`) bearbeiten, so dass sie in einem Shellscript oder einer Batch-Datei abgelegt werden können, aber auch dort wäre ein langer Ausdruck nicht optimal für die Lesbarkeit und Wartbarkeit. Darum definiere ich in den Zeilen 26 bis 38 Funktionen, die das Testen der Markierungen in der Stelle `state` vereinfachen sollen. In den Funktionen kann jetzt auch auf die Namen des *enum* Typs `state` zugegriffen werden.

Es ist auch wichtig zu bemerken, dass MARIA die Definition des Netzes lediglich einmal einliest. Darum müssen die Funktionen hier nach der Definition der Stelle, auf die sie zugreifen, definiert werden.

Das wichtigste ist aber, dass das Netz fair und lebendig sein soll, darum wird (in Anlehnung an die Modifikation in [Valk und Moldt, 2006, S.88f]) eine Stelle `plates`

eingeführt, deren Anfangsmarkierung jeweils eine Marke weniger enthält, als Philosophen vorhanden sind (bildlich gesprochen nimmt sich jeder Philosoph mit der linken Gabel zusätzlich einen Teller, den er nach dem Essen auch wieder zurücklegt.) Die modifizierten Stellen und Transitionen finden sich in den Zeilen 9 bis 24.

Zusätzlich wurde der Modelchecker in Zeile 16 angewiesen, nur Zustandsübergänge zu berücksichtigen, die alle Transitionsinstanzen der Transition `left` fair behandeln.

Nun lässt sich die Anfrage

```
<>(cardinality subset st {place state} (st.s==2)) > 0
```

verständlicher (wenn auch etwas berechnungsaufwändiger) mit dem Existenzquantor formulieren als

```
<> philosopher pp || iseating(pp)
```

d. h. es gilt in allen möglichen Pfaden in irgendeinem Zustand, dass es einen Philosophen `pp` gibt, welcher im Zustand `eating` befindlich ist. Mit dem Allquantor lässt sich prüfen, ob alle Philosophen in allen Pfaden etwas zu essen erhalten

```
philosopher pp && <> iseating(pp)
```

und es gilt sogar, dass alle Philosophen immer wieder etwas zu essen bekommen

```
philosopher pp && []<> iseating(pp)
```

was aber nicht bedeutet, dass sie die ganze Zeit essen:

```
philosopher pp && [] iseating(pp)
```

Funktionen lassen sich auch an der Kommandozeile von `maria` definieren, mit dem vorangestellten Schlüsselwort *`function`*:

```
@0$function unsigned numeating = cardinality philosopher pp (iseating(pp)): pp
@0$visual path numeating > 1
```

(In diesem Fall wird eine Multimenge spezifiziert, die alle Philosophen enthält, die im derzeitigen Zustand essen und die Mächtigkeit dieser Menge zurückgegeben.)

Mit den hier gegebenen Beispielen werden einige wichtige Möglichkeiten von MARIA erläutert, aber es werden in Abschnitt 6.5 (S.24) auch daraus resultierende Problemquellen ersichtlich werden.

4 Vergleich der Petrinetzformalismen von RENEW und MARIA

In diesem Abschnitt soll es darum gehen, ob sich die zugrunde liegenden Formalismen der Programme RENEW und MARIA ähnlich genug sind, um eine Übersetzung praktikabel und sinnvoll erscheinen zu lassen.

Beim Vergleich zwischen Java-Referenznetzen und der konkreten Ausprägung algebraischer Systemnetze, die in MARIA verwendet werden kann, fallen die folgenden Unterschiede auf:

4.1 Es gibt im Unterschied zu RENEW keine Netzexemplare in MARIA.

Dies ist ein fundamentaler Unterschied. Es gibt auch nichts vergleichbares.

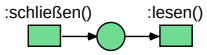

Abbildung 3: Das Netz „`logbuch`“

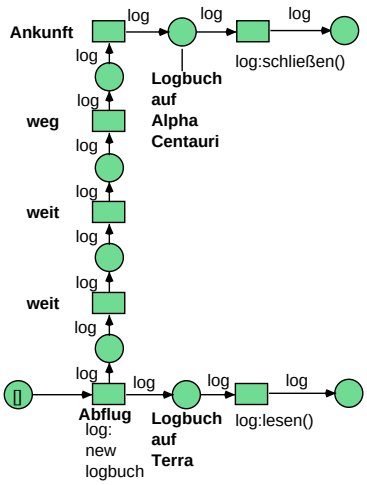

Abbildung 4: RENEW konformes Alpha Centauri Beispiel

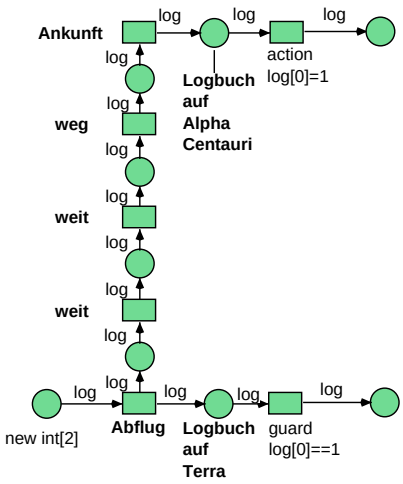

Abbildung 5: Centauri Beispiel mit Logbuch als Java-Objekt

4.2 Seiteneffekte und Lokalität

Es gibt in MARIA keine (unbeschränkten und polymorphen) Zeiger oder Referenzen und keinerlei Zuweisungen (abgesehen von Markierungsänderungen durch Transitionen). Seiteneffekte und nicht-lokale Auswirkungen sind durch die Sprachdefinition der Anschriftsprache *vollständig ausgeschlossen*, das Lokalitätsprinzip bei Petrinetzen wird streng eingehalten, ganz anders als bei RENEW.

Weil dies offenbar ein strittiger Punkt ist, der aber für die Anwendbarkeit von optimierten Model Checking Algorithmen auf Referenznetze bedeutsam ist, folgen einige Argumente.

Das Prinzip der Lokalität für Petrinetze wird in [Valk und Moldt, 2006, S.40] folgendermaßen definiert:

Lokalität **Definition 6.** ***Prinzip der Lokalität für Petrinetze****: Das Verhalten einer Transition wird ausschließlich durch ihre Lokalität bestimmt, welche sich aus ihr und der Gesamtheit ihrer Eingangs- und Ausgangs-Elemente zusammensetzt.*

Zuweisungen können in RENEW zwar direkt nur in den speziellen `action` Inschriften überhaupt verwendet werden, so dass der Anwender auf Grund des verwendeten `action`-Schlüsselworts einen Hinweis auf mögliche nicht-lokale Auswirkungen erhält.

Allerdings gibt es z. B. für Berechnungen mit Arrays oder Feldern eines Objekts keine Alternative zu `action` Inschriften, anders als in MARIA, wo Idiome zur Berechnung von Ausdrücken aus Feld- und Struct- Werten zur Verfügung stehen, so dass diese Datentypen ohne Seiteneffekte an Transitionen und Kanten verwendet werden können. Der Ausdruck `a.{b c}` liefert z. B. ein *`struct`*, dessen Wert sich aus der *`struct`*

Variable `a` berechnet, dessen Komponente `b` aber den Wert `c` erhält. Analog dazu liefert der Ausdruck `a.{[b] c}` den am Index `b` modifizierten Wert der Array-Variablen `a` [Mäkelä u. a., 2005, S.21f].

In RENEW wiederum sind Zuweisungen und Seiteneffekte generell zwar außerhalb von `action`-Inschriften und der Bindung von neuen Netzexemplaren an Bezeichner unerwünscht, weswegen alle Zuweisungsoperatoren explizit aus der Anschriftsprache entfernt wurden.

Allerdings werden Seiteneffekte durch das Fehlen von Zuweisungsoperatoren außerhalb von `action` Anschriften nicht vollständig ausgeschlossen, sondern können weiterhin z. B. durch Methodenaufrufe auftreten, mit zum Teil komplexen (und möglicherweise unerwünschten) Auswirkungen (vgl. [Kummer u. a., 2006b, S.57]).

In Abbildung 5 auf der vorigen Seite ist Analog zum Beispiel in [Kummer, 2002, S.36] (Abbildung 4 links gegenüberliegend) ein Java-Referenznetz dargestellt, welches in einer Transition mit einer `action` Anschrift ein Array modifiziert. Das Netz in Abbildung 5 funktioniert in RENEW allerdings nur, wenn das Lesen des Logbuchs (die Transition mit dem `guard`-Ausdruck) auf Terra manuell ausgelöst wird, weil der Simulator Zustandsänderungen an Java-Objekten in Markierungen nicht automatisch berücksichtigt. Diese Beschränkung des Simulators lässt sich allerdings in Netzen auch maskieren, wie das Netz in Abbildung 6 zeigt. Hier ist im Gegensatz zum gutartigen Verhalten des Alpha Centauri Netzes ein Schreib-Lese Konflikt[7] gezeigt. Bei diesem Netz müssen im RENEW-Simulator keine Transitionen manuell geschaltet werden[8], das Verhalten entspricht also dem der Netzreferenzen in Abbildung 4.

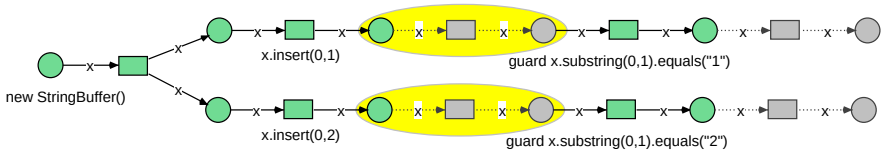

Abbildung 6: Ein Beispielnetz für nicht-lokale Auswirkungen von Transitionen mit Methodenaufrufen

Also können im konkreten Fall bei Java-Referenznetzen Effekte auftreten, die nicht der klassischen Definition des Lokalitätsprinzips entsprechen und die im Verhalten des Netzes zu Problemen führen können.

Eine Erörterung der Frage, ob Netzreferenzen im Formalismus der Referenznetze selbst das Lokalitätsprinzip der Petrinetze verletzen ist in [Kummer, 2002, S.36-38] zu finden.

Bezug nehmend auf [Valk, 2000] argumentiert Kummer, dass zwar durch das Schreiben des Logbuchs auf Alpha Centauri eine Transition auf Terra aktiviert wird, aber sich der Inhalt der Stelle im Vorbereich der Transition nicht ändert. Das ist insofern richtig, als die Referenz immer noch auf das gleiche Netz zeigt, allerdings nimmt Definition 6 auf Seite 16 ausdrücklich auf das *Verhalten* der Transition Bezug, das sich in diesem

[7] und zwar zur Demonstration ohne `action` Inschriften - ein Netz mit entsprechendem Verhalten lässt sich mittels `action` Anschriften leicht auch mit Java-Arrays konstruieren -

[8] Das ändert sich, sobald man die Vergleiche an den Guards austauscht. Dann wird es entscheidend, wann die Transitionen in den mit Ellipsen hinterlegten Bereichen schalten. Wenn zusätzlich noch die elliptischen Bereiche durch jeweils eine einzige Stelle vergröbert werden, können bei bestimmten Schaltfolgen in den `guards` gar keine Bindungen mehr automatisch gefunden werden.

Fall durch das Schalten einer entfernten Transition ändert.

Was heißt also „Lokalität“ bei Referenznetzen? Im Zusammenhang des Zitats der Definition 6 in [Valk und Moldt, 2006, S.43] ist mit Lokalität formal die Vereinigung von Vor- und Nachbereich eines Knotens gemeint, im Falle von Transitionen besteht ihre Lokalität also aus Stellen. Allerdings wird zwar Lokalität formal definiert, aber keine formale Definition des Lokalitätsprinzips gegeben. In [Girault und Valk, 2003, S.12] findet sich die gleiche Definition und es wird später festgestellt, dass das Lokalitätsprinzip für gefärbte Netze erhalten bleibt und dort die Lokalität einer Transition „als alle Stellen, Kanten und Guards, die mit dieser Transition verbunden sind“ verstanden werden sollte [Girault und Valk, 2003, S.38]. Das heißt, das Verständnis des Lokalitätsbegriffs wird für gefärbte Netze um die (hinzu getretenen) Guards erweitert.

Eine Definition des Lokalitätsbegriffs für Referenznetze habe ich in der von mir gesichteten Literatur nicht finden können.

Möglicherweise besteht aber ein Ausweg darin, den Lokalitätsbegriff für Referenznetze entsprechend anzupassen.

In [Kummer, 2002, S.36-38] wird im Zusammenhang mit dem Alpha Centauri Beispiel jedenfalls von „scheinbar nichtlokaler Wirkung“ gesprochen, was zu implizieren scheint, dass alle in dem Netz in Abbildung 4 (S.16) auftretenden Effekte lokal sind.

Ein denkbarer *Versuch* wäre also wie folgt:

Lokalität einer Transition in einem Referenznetz-exemplar

Definition 7. *Die **Lokalität einer Transition in einem Referenznetzexemplar** besteht*

1. *aus allen Stellen, Kanten und Guards, die mit dieser Transition verbunden sind,*

2. *sowie aus allen Stellen, Kanten und Guards die mit Transitionen verbunden sind, mit denen die Transition einen synchronen Kanal hat,*

3. *sowie aus allen Stellen, in denen Referenzen auf Netzexemplare liegen, die bereits durch Netzreferenzen in der in den Punkten 1 bis 2 spezifizierten Lokalität referenziert werden.*

Der letzte Punkt ist notwendig, weil Netzreferenzen auch in Stellen liegen können, die nicht mit der aktuellen Transition durch synchrone Kanäle verbunden sind, was bedeuten könnte, dass ein etwaiger Model Checking Algorithmus zu jedem Netzexemplar eine Liste von Referenzen führen müsste.

In diesem etwas sperrigen Definitionsversuch sind noch keine Java-Objekte berücksichtigt, allerdings ist es sehr zweifelhaft ob z. B. Methodenaufrufe die Netzwerkkommunikation oder Eingriffe in den Zustand des Simulators zur Folge haben können (siehe dazu auch [Duvigneau, 2002, S.126]), wirklich unter Lokalität gefasst werden sollen.

Es zeigt sich, dass es viele Quellen von Seiteneffekten in Referenznetzen mit Java als Anschriftsprache gibt. Deren Auswirkungen müssten bei einer Integration von Model Checking Verfahren in RENEW in jedem Falle berücksichtigt werden. In Abschnitt 5.1 (S.20) wird auf diese Problematik im Zusammenhang mit den Ausführungen zum nächsten Punkt zurückgekommen.

4.3 Es gibt keine synchronen Kanäle in MARIA.

Allerdings können Netze in MARIA modular aufgebaut sein, d. h. aus einer baumartigen Hierarchie mehrerer Subnetze bestehen. Das Hauptnetz ist an der Wurzel des Baums und alle seine Transitionen sind im globalen Zustandsraum sichtbar. Interne Transitionen eines Subnetzes modellieren interne Aktionen und Kommunikation zwischen den Teilnetzen kann durch Verschmelzen von Transitionen geschehen (nach Mäkelä u. a. [2005]).

Ähnlich wie bei synchronen Kanälen in Referenznetzen schalten die verschmolzenen Transitionen synchron, allerdings scheint die Semantik der Aktivierung bei den verschmolzenen (auch: synchronisierten) Transitionen anders intendiert zu sein als bei synchronen Kanälen. In [Mäkelä, 2003] schreibt Mäkelä:

> The fused transition t can only be enabled if all its components in the child nets are enabled in some marking M^* reachable from M_1 via a possibly empty sequence of internal transitions.

(wobei M_1 die aktuelle Markierung ist).

Bei synchronen Kanälen hingegen wird gefordert, dass alle an der Kommunikation teilnehmenden Transitionen in der *aktuellen* Markierung aktiviert sein sollen.

Der feine Unterschied in der Semantik ist durch die intendierte Anwendung motiviert. [Kummer, 2002] beruft sich auf die Arbeit [Christensen und Hansen, 1994], in der primär um Synchronisation und synchrone Kommunikation geht. In [Mäkelä, 2003] wird die Arbeit [Christensen und Petrucci, 1995] zitiert, die direkt auf Zustandsraum-Analyse Bezug nimmt. Die Tatsache, dass an beiden zugrunde gelegten Arbeiten maßgeblich der gleiche Autor beteiligt war, und sie annähernd zur gleichen Zeit entstanden, dürfte aber auch die Ähnlichkeiten in den von Kummer und Mäkelä darauf aufbauend entwickelten Formalismen erklären.

4.4 Auswertung

Die Unterschiede in den Netzsemantiken der beiden Programme lassen sich natürlich auf den intendierten Anwendungsbereich zurückführen. In RENEW war von vornherein das Hauptaugenmerk auf Simulation gerichtet und „Analyse blieb fast vollständig unberücksichtigt“ [Kummer, 2002, S.233], auch lassen sich mit RENEW tatsächlich komplette Anwendungen programmieren, während MARIA sich besser zur Verifikation von Modellen eignet (insbesondere lassen sich mit einem Übersetzer SDL2PN (Protokoll-)Spezifikationen in der Sprache SDL direkt in MARIA Netze übersetzen, siehe [Aalto, 2004]). Im Gegensatz dazu wurde RENEW als eine Art Entwicklungsumgebung für Petrinetze konzipiert. Die Netze werden in der Simulation getestet und können mit Funktionen untersucht werden, die denen eines Debuggers entsprechen.

Alle oben genannten Unterschiede machen es unmöglich, Java-Referenznetze in die von dem Tool MARIA verwendete Sprache direkt zu übersetzen oder sie in irgendeiner Form mit diesem Model-Checker zu prüfen. Darüber hinaus machen die diversen in RENEW möglichen Seiteneffekte Probleme bei den in MARIA verwendeten Model Checking Verfahren.

Für eine Übersetzung zwischen den beiden Formalismen wäre es also entweder nötig, beide stark einzuschränken, oder einen der beiden Formalismen entsprechend zu erweitern.

Im ersten Falle würde der der Nutzeffekt stark unter der Einschränkung leiden, weil sowohl die Mächtigkeit der synchronen Kanäle der Netzexemplare und der Java-Anschriften in RENEW als auch die gute Unterstützung der Zustandsraum-Analyse in MARIA verloren ginge.

Im zweiten Fall stünde der Nutzeffekt in keinem günstigen Verhältnis zum Aufwand.

In beiden Fällen wären starke Eingriffe in den Formalismus und die Software notwendig. Und wenn schon einer der beiden Formalismen erweitert werden soll, sollte dies schon der von RENEW sein, womit wieder die direkte Integration von Model Checking in RENEW nahe liegender wäre, als eine Anpassung an einen fremden Formalismus.

Zusammenfassend erscheint eine Angleichung oder Übersetzung der beiden Formalismen nicht vielversprechend.

Möglicherweise ergibt sich aber aus der Ähnlichkeit der Formalismen der hierarchischen Subnetze und der synchronen Kanäle ein Ansatz für ein für Referenznetze brauchbares Model Checking Verfahren.

Im Abschnitt 5 werde ich einige Ansätze zur Integration von Model Checking in RENEW diskutieren.

5 Alternativen: Wie kann Model Checking in RENEW integriert werden?

In diesem Abschnitt werden Alternativen zur Integration von Model Checking in RENEW diskutiert. Dabei erscheint die erste Variante, Model Checking für Referenznetze, zum Einen als optimal, zum Anderen aber gibt es Gründe warum sie mindestens sehr aufwändig und darum im Rahmen dieser Arbeit nicht leistbar ist; auch ist ungeklärt, in wie weit sie überhaupt praktikabel ist. Dann werden kurz zwei Vereinfachungen untersucht, die aber wenig interessant erscheinen, um zum Schluss den gewählten Ansatz als bewusst gewählten Kompromiss darzustellen.

5.1 Model Checking für Referenznetze

Eine dichte Integration von Model Checking in RENEW, in einer Weise, die den gesamten Formalismus unterstützt, wäre eine äußerst attraktive, aber leider auch zur Zeit ausgesprochen utopische Unternehmung. Im Folgenden werden einige Gründe für diese Einschätzung gegeben.

Bereits im Abschnitt 4 (S.15) wurde ein wichtiger Unterschied zwischen den Formalismen MARIAs und RENEWs deutlich: In Java-Referenznetzen gibt es Arten von Seiteneffekten, die in MARIA nicht möglich sind: die Veränderung von Markenobjekten in entfernten Stellen.

Diese Effekte beruhen fundamental auf der Referenzsemantik.

Es wurde festgestellt, dass in MARIA spezielle Vorkehrungen getroffen wurden, um Seiteneffekte zu vermeiden, die den verwendeten Model Checking Verfahren zuwider laufen würden.

Ein Model Checking Algorithmus für Referenznetze müsste diese Effekte entweder berücksichtigen oder Methodenaufrufe und Action-Inschriften ebenso konsequent

verbieten, wie jetzt Zuweisungen verboten sind. Die so prüfbaren Netze wären zwar immer noch Referenznetze, aber nicht mehr mit der vollen Ausdrucksmächtigkeit von RENEW, also mit deutlich eingeschränkter Nützlichkeit. Java ohne jede Form der Methodenaufrufe wäre in der Tat eine sehr starke Einschränkung. Alternativ könnte man versuchen, Methodenaufrufe mit Seiteneffekten zu verbieten, aber es ist vollkommen unklar, wie das funktionieren sollte. Mir ist keine Methode bekannt, wie man in Java auch nur *feststellen* kann, ob ein Methodenaufruf einen Seiteneffekt haben wird. Dies gilt natürlich ebenso für die meisten anderen Programmiersprachen, mit einigen wenigen Ausnahmen, wie z. B. Haskell.

Selbst bei Einschränkung auf eine Seiteneffekt-freie Anschriftsprache bleibt das Problem der synchronen Kanäle und der Referenzen auf Netzexemplare.

In [Christensen und Petrucci, 1995] wird ein Verfahren beschrieben, mit dem für Modulare Gefärbte Netze jeweils einzelne Erreichbarkeitsgraphen für die Teilnetze und Synchronisationsgraphen für die verschmolzenen Transitionen erstellt werden können. Dadurch wird vermieden, dass für ein dem Modularen Gefärbten Netz äquivalentes Gefärbtes Netz der (wesentlich größere) Erreichbarkeitsgraph gebildet werden muss. Das von MARIA verwendete Verfahren baut u. a. auf dieser Technik auf ([Latvala, 2001]).

Um die Zustandsexplosion für RENEW Netze einzuschränken, könnte aufbauend auf den Forschungen von Christensen, Damgaard Hansen, Latvala, Petrucci und anderen ein Verfahren zur Zustandsraumberechnung entwickelt werden, das mindestens Netzreferenzen und möglicherweise sogar Java-Objekte berücksichtigt.

Für die Behandlung von Java-Objekten wären möglicherweise Ansätze aus [JavaPathFinder, 2006] interessant, soweit die Lizenz es erlaubt.

Auf jeden Fall wäre ein solches Projekt nicht in einer Studienarbeit zu leisten und nach meinem bisherigen Eindruck ist es auch sehr zweifelhaft, ob die nötige Grundlagenforschung im Rahmen einer Diplomarbeit geleistet werden kann.

Selbst wenn dieses Vorhaben angegangen wird, bleibt die Frage, ob die verfügbaren Optimierungsverfahren ausreichen, um komplexe RENEW-Anwendungen wie etwa die Agentenplattform mit Model Checking zu verifizieren. Der aktuelle Stand der Technik im Jahre 2006 legt die Vermutung nahe, dass sich das so entwickelte Verfahren zunächst nur zu Lehrzwecken für kleinere Modelle nutzen ließe. Diese Einschätzung lässt sich untermauern, wenn man einmal versucht, eins der komplexeren Beispielnetze aus den MARIA Quellen, wie etwa `resource.pn` komplett auszufalten, was je nach verfügbarer Rechenleistung seine Zeit dauert.

5.2 Vereinfachte Netze

Alternativ zum vollen Java-Formalismus ließe sich entweder einer der einfacheren zur Verfügung stehenden Formalismen mit Model Checking bearbeiten, wie etwa S/T-Netze, oder aber eine Reduktion eines bestehenden Formalismus.

5.2.1 Verifikation eines einfachen Formalismus

Model-Checking von S/T-Netzen (in RENEW: „P/T-Nets“) oder auch „Bool-Nets“ wäre ein durchaus gangbarer Weg. Da es gute Verfahren für gefärbte Netze gibt und

sich diese leicht auf einfachere Formalismen abbilden lassen, sollte der Forschungsaufwand sehr begrenzt sein, allerdings kann der Implementationsaufwand immer noch als beträchtlich eingeschätzt werden, sofern, wenn auch vereinfachte, Model Checking Algorithmen in RENEW integriert werden sollten.

Allerdings könnten hier auf Grund der geringen zu erwartenden Komplexität der zu prüfenden Netze auch naive Verfahren verwendet werden, die sich mit nur wenig Optimierungen begnügen.

Auf Grund der Beschränkung der Formalismen wäre die Implementation jedoch vermutlich nur zu Lehrzwecken nützlich. Dann allerdings wäre es sogar interessant und notwendig, die Auswirkungen verschiedener Optimierungen an der Größe des generierten Zustandsraums, Speicher- und Zeitbedarf und anderen Parametern zu beobachten.

Alternativ könnten S/T-Netze auch einfach in die Netzbeschreibungssprache eines gebräuchlichen Model Checking-Tools übersetzt werden.

5.2.2 Abstraktion von Netzeigenschaften

Wie bereits in den Abschnitten 5.1 (S.20) und 4 (S.15) erläutert wurde, gibt es grundsätzliche Probleme bei der Anwendung von optimierten Model Checking Verfahren auf Referenznetze. Wenn also von Netzeigenschaften abstrahiert wird, um die Netze für bekannte Verfahren handhabbar zu machen, müsste gerade auch von sehr grundlegenden Eigenschaften, wie synchronen Kanälen, Java-Anschriften (mindestens Methodenaufrufen und `action` Inschriften) und Netzexemplaren und -referenzen abstrahiert werden.

Übrig bliebe eine Art Formalismus für gefärbte Netze, der nur die Grundtypen von Java sinnvoll unterstützt, da die Anwendung von Objekten ohne Methodenaufrufe wenig nutzbringend erscheint. Insgesamt scheint ein solcher Formalismus wenig interessant. Er ließe sich zwar sehr einfach in die Netzsprache von MARIA übersetzen, aber dann fragt sich, warum nicht gleich die wesentlich mächtigeren Möglichkeiten MARIAs ausgenutzt werden sollen.

6 Der MARIA-Formalismus für RENEW

In Abschnitt 4 (S.15) wurde die Möglichkeit einer Übersetzung zwischen MARIA- und RENEW-Netzen ausgeschlossen und einige Probleme aufgezeigt, derentwegen in Unterabschnitt 5.1 (S.20) und den folgenden das Projekt einer direkten Integration von Model-Checking Verfahren in RENEW vorerst beiseite geschoben wurde.

Nachdem alle Ideen einer direkten Integration von Model Checking in RENEW ausgeschlossen wurden, bleibt die Möglichkeit, einen Formalismus zu schaffen, in dem Petrinetze mit den MARIA-eigenen Anschriften in RENEW grafisch spezifiziert werden können und darüber hinaus von RENEW aus das Kommando `maria` fernzusteuern.

Die Grundidee dabei ist, dass alle Aufgaben die das MARIA-Tool übernehmen kann, auch von diesem übernommen werden sollen. RENEW wird beinahe, aber nicht ganz zu einem Malprogramm für Petrinetze degradiert, nicht ganz, denn es zeigt sich dass MARIA keinen Befehl zur Syntaxprüfung partieller Netzanschriften zur Verfügung stellt. Diese Funktionalität muss also RENEW anbieten.

Außerdem bietet sich die Möglichkeit einer späteren Integration in den Simulator.

6.1 Grundidee

Der Anwender malt also wie gewohnt Transitionen, Stellen und Kanten und versieht diese mit Anschriften. Die Anschriften werden, wenn der MARIA-Formalismus gewählt wurde, sofort auf syntaktische Korrektheit überprüft. Das so erhaltene Netz kann entweder explizit als MARIA-Netz mit der Erweiterung „`.pn`“ abgespeichert und von Hand in MARIA geprüft werden, oder RENEW tritt nach Aufruf eines entsprechenden Menüpunktes in Interaktion mit dem Programm `maria` und zeigt die Ergebnisse von MARIA-Befehlen an.

Grundsätzlich wäre auf diese Weise auch eine Integration in den Simulator möglich, denn MARIA erlaubt es dem Benutzer, mit Hilfe der Befehle *`succ`* und *`go`*, sich frei im Zustandsraum zu bewegen, zeigt die jeweils aktivierten Bindungen an und erlaubt mit dem Kommando *`show`* eine Ausgabe der aktuellen Markierungen. Die so erhaltene Ausgabe könnte eingelesen und interpretiert und anschließend im Simulator grafisch dargestellt werden. In dieser Arbeit wird lediglich auf Grund des Implementationsaufwandes und dem geringen zusätzlichen Nutzen darauf verzichtet. Im grafischen Browser kann mit dem Kommando *`visual show`* ohnehin der Zustandsraum auch interaktiv untersucht werden und vermutlich wäre die Simulation über diesen Umweg auch relativ träge, weil auf die Ausgabe und das Parsing gewartet werden müsste.

Offenbar gab es bereits früher einen Versuch von Olaf Kummer[9], MARIA Netze in RENEW zu erstellen und zu exportieren, in Form eines Modus für Maria Netze, der aber nicht an RENEWs aktuelle Plugin-Architektur angepasst wurde. Die im `Gui`-Baum der RENEW-Quellen bereits vorhandenen Klassen dieses Modus waren bei der Entwicklung des in dieser Arbeit beschriebenen `MariaNets` Plugins zum Teil sehr hilfreich.

6.2 Netzanschriften für MARIA

Zunächst soll mehr als ein Deklarationsknoten erlaubt werden, damit Anweisungen an den Model Checker und andere Netzanschriftstypen zum einen grafisch von anderen Deklarationen abgesetzt und zum anderen auch beim Export an die richtige Stelle im generierten Text gestellt werden können.

Für die sofortige Syntaxprüfung ist in der Terminologie von RENEW ein Compiler nötig (siehe Abschnitt 2.3 (S.6)), der alle in MARIA möglichen Netzanschriften prüfen kann, aber keine Übersetzung in Java-Objekte vornehmen muss. Der Compiler kann wie alle anderen Netzformalismen ausgewählt werden, allerdings wird zunächst beim Versuch, ein MARIA-Netz zu simulieren, ein Fehler ausgegeben, bedingt dadurch dass der Compiler keine Objekte generiert. Es wäre möglich, statt des Simulators den Model-Checker zu starten, aber das würde einer späteren Integration in den Simulator im Wege stehen.

Weiterhin soll es möglich sein, Anschriften aufzuspalten und für den Export zusammenzufügen, weil diese recht komplex werden können und dem Anwender somit größere Freiheit bei der übersichtlichen Gestaltung seines Netzes gelassen wird. Dieses

[9] persönliche E-Mail an den Autor

Verfahren wurde bereits in der Klasse `de.renew.gui.maria.MariaExporter` von Olaf Kummer implementiert und ist von mir auf die aktuelle MARIA-Version angepasst und erweitert worden. In Abschnitt 6.5 werden Probleme mit dieser Aufspaltung von Anschriften angesprochen.

6.3 Der Anfrageknoten „Query Node“

Damit nicht jedes Mal die gleichen Befehle an der MARIA-Shell eingegeben werden müssen, soll der Benutzer diese zusammen mit dem Netz abspeichern können, dafür wird ein neuer Anschriftsknoten für RENEW Netze entwickelt, der „Query Node“ heißen soll und vom Deklarationsknoten („Declaration Node“) abgeleitet ist.

Die Befehle der MARIA-Shell werden in den Parser integriert, so dass auch die Abfragen im „Query Node“ einer sofortigen Syntaxprüfung unterliegen.

6.4 Kommunikation mit `maria`

Es gibt verschiedene Alternativen, wie Kommunikation zwischen Programmen erfolgen kann, allerdings sind nur wenige davon möglich, ohne dass das jeweilige Programm daran speziell angepasst wird und gleichzeitig portabel über mehrere Architekturen.

Kommunikation über ein TCP/IP Socket scheidet aus mehreren Gründen aus: Die Kommunikation über eine TCP/IP fähige Wrapper-Anwendung löst keine Probleme. Denn entweder müsste dann das Problem geklärt werden, wie der Wrapper mit MARIA kommuniziert, oder wenn ein fertiges Programm, wie z. B. `inetd` verwendet wird, besteht die Frage, wie RENEW dann mit diesem Wrapper (der nebenbei auch nicht portabel ist) kommuniziert. Also müsste MARIA in diesem Fall an TCP/IP-Kommunikation angepasst werden. Das wiederum würde die Frage aufwerfen, wie sich RENEW an der TCP/IP fähigen MARIA Version authentifizieren sollte, wenn keine Sicherheitslücken entstehen sollen. Außerdem müssten etwaige Anwender in Kauf nehmen, einen Patch zu applizieren und MARIA selbst zu übersetzen.

Unix-Domain Sockets lösen das Problem der Authentisierung, scheiden aber aus Portabilitätsgründen aus. Ähnliches gilt für (Named) Pipes.

Bleibt die Kommunikation über Ein- und Ausgabestrom, die von MARIA konzeptionell unterstützt wird und die in Java mit den Klassen im `java.io` Paket verwendet werden kann. Diese Technik steht laut Dokumentation der entsprechenden Klassen auf allen Architekturen zur Verfügung, für die Java zur Verfügung steht. Ein erfolgreiches Experiment im Vorlauf dieser Arbeit hatte eine wiederverwendbare Klasse (`ProcessConsoleTalker.java`) zum Ergebnis, die über Ein- und Ausgabestrom mit einem Prozess kommunizieren und als Grundlage für die Kommunikation von RENEW mit MARIA dienen kann.

6.5 Auslassungen und Probleme

MARIA ist ein sehr komplexes Programm und es gibt zwei Features, die sich nicht einfach mit grafischen Netzelementen und Anschriften ausdrücken lassen.

Preprozessor-Direktiven Zum Einen können MARIA Netzbeschreibungen mit Direktiven versehen werden, die dem C-Preprozessor sehr ähnlich sind, so dass die gleiche Datei mit Hilfe von `#ifdef`, `#include` und ähnlichen Konstruktionen sehr unterschiedliche Netze erzeugen kann. Wie solche Direktiven grafisch spezifiziert werden können, ist nicht ersichtlich, darum wurden sie weggelassen.

Modulare Netze Zum Anderen erlaubt es MARIA, modulare Netze mit mehreren Subnetzen zu spezifizieren. Ebenso wie bei den Preprozessor-Direktiven ist hier nicht auf Anhieb offensichtlich, wie diese mit Hilfe des grafischen Editors spezifiziert werden können. Damit ist leider ein sehr leistungsfähiges Feature MARIAs von RENEW aus nicht verwendbar. Denkbar wäre die Implementation von MARIA Subnetzen über mehrere RENEW Zeichnungen mit gleichem Namenspräfix, die im N-zu-1 Verfahren exportiert werden müssen, aber die genaue Ausführung ist noch unklar.

Abhängigkeiten zwischen Bezeichnern Ein grundsätzliches Problem besteht auch darin, dass die Netzbeschreibungen, wie bereits in Abschnitt 3.4 (S.13) erwähnt, nur ein einziges Mal von MARIA eingelesen werden, so dass *alle* Bezeichner textuell vor ihrer Verwendung definiert sein müssen. Das wird in der Praxis zu Problemen führen, wenn z. B. beim Export nicht erkannt werden kann, auf welche Bezeichner eine Funktion zugreift und an welcher Stelle sie verwendet wird und demzufolge nicht entschieden werden kann, an welcher Stelle in der Datei die Definition platziert werden muss.

Auch hierfür war eine generelle Lösung zunächst nicht offensichtlich, aber trotzdem können immer noch Funktionen in Transitionsdefinitionen spezifiziert werden, beim Export werden diese dann zwischen die eingehenden und die ausgehenden Kanten geschrieben, da es möglich ist, dass in den eingehenden Kanten Variablen gebunden werden, die in der Funktion benötigt werden. Diese Entscheidung führt dazu, dass jedenfalls alle nicht-modularen Beispiele in den Maria Quellen mit dem neuen Plugin erzeugt werden können. Mit einer gewissen Vorsicht können Funktionen auch in Deklarationsknoten verwendet werden, mit dem Effekt, dass sie *vor* allen anderen Netzelementen erscheinen. (Also können in diesen Funktionen keine Markierungen von Stellen referenziert werden.)

Unproblematisch ist dagegen die Verwendung von Funktionen im Query Node. Im Gegensatz zum Beispiel in Listing 2 (S.14) *müssen* die Helferfunktionen im Maria Formalismus für RENEW im Query Node spezifiziert werden, aber genau dort werden sie ja auch benötigt.

Kurz entschlossen habe ich kurz vor Fertigstellung der Arbeit noch einen Importfilter für Maria Netze geschrieben, mit dessen Hilfe ich das sehr komplexe Beispiel des Sliding Window-Teilprotokolls des HDLC Protokolls konvertiert und getestet habe.

Bei besagtem Netz (`swn.pn` im Verzeichnis `parser/test` in den MARIA Quellen) gibt es ein besonderes Problem: In der Transition `rec_data` ist die Reihenfolge der ausgehenden Kanten relevant, weil in der ersten ausgehenden Kante die Variable `data` gebunden wird, die in der letzten ausgehenden Kante benötigt wird. Ob dieses Netz nach dem Export funktioniert, hängt von der Reihenfolge der Abarbeitung der Kanten beim Export ab, auf die z. Z. keine Kontrolle ausgeübt wird.

Eine generelle Lösung für diese Art von Abhängigkeiten wird in der Beschreibung

der Quelldatei `MariaCreator.java` (S. 31) diskutiert. In der aktuellen Version ist dieser Ansatz zwar nur für initiale Markierungen von Stellen implementiert, ließe sich aber auf alle hier beschriebenen Abhängigkeitsprobleme verallgemeinern.

Natürlich gelten die in diesem Abschnitt beschriebenen Einschränkungen auch für den Import.

Weitere Einschränkungen: Preprozessor-Direktiven werden beim Import vollständig ignoriert, was zur Folge hat, dass alle (auch widersprüchlichen) Teile der Beschreibung eingelesen werden und `#include` Direktiven zu Fehlermeldungen (unbekannte Bezeichner) in MARIA führen. Modulare Netze führen dazu, dass beim Import eine `ParseException` geworfen wird, da die Syntax für Subnetze und verschmolzene Transitionen dem Parser unbekannt ist.

7 Implementation des `MariaNets` Moduls

Nachdem die konzeptionellen Entscheidungen im vorigen Abschnitt beschrieben wurden, sollen nun einige konkrete Eigenschaften der Implementation erklärt werden. In Abschnitt 8 (S.38) wird außerdem ein kurzes Anwendungsbeispiel gegeben, das allerdings sowohl gute Kenntnisse der Benutzung von RENEW als auch MARIA voraussetzt.

7.1 Einbindung in die RENEW-Architektur

Das Plugin erhält den Namen `MariaNets`. Da zu Beginn der Arbeit unklar war, in wie weit das Endergebnis brauchbar sein würde und zum RENEW Konzept passt, habe ich auf einem privaten Quellverzeichnis außerhalb des CVS Baums gearbeitet und dort auch nur eine Quelldatei[10] außerhalb des `MariaNets` Baumes verändert. Die Steuerdateien `plugin.cfg` und `build.xml` wurden jedoch so angelegt, dass eine Integration in die offiziellen Quellen einfach erfolgen kann.

Aus dem gleichen Grund habe ich als Pakethierarchie `de.uni_hamburg.tgi.renew.marianets` gewählt.

Die Integration des Plugins erfolgt wie in Abschnitt 2.2 (S.6) beschrieben.

Das Plugin erfüllt drei Hauptfunktionen

1. Parsing von Maria Netzbeschreibungen und Anschriften.

   Sowohl zur Syntaxprüfung von Anschriften als auch zum Import von Maria Netzen ist hierbei in Abschnitt 7.2.2 (S.28) beschriebene Parser `MariaParser.jj` von zentraler Bedeutung.

2. Erzeugen von Netzbeschreibungsdateien für MARIA aus RENEW Netzen mit entsprechenden Anschriften.

   Die zentrale Funktionalität liegt hierfür in der in Abschnitt 7.2.3 (S.31) beschriebenen Klasse `MariaCreator`.

[10] `CH.ifa.draw.util.StorableInput` musste angepasst werden, da gespeicherte Objekte mit einem Unterstrich („_“) im Paketnamen („`uni_hamburg`“) nicht eingelesen werden konnten.

3. Kommunikation mit einem laufenden MARIA Prozess.

   Mehrere Klassen teilen sich hier die wesentlichen Aufgaben der Kommunikation und der Darstellung der Ergebnisse.

7.2 Beschreibung der Quelldateien und Klassen

7.2.1 `MariaNetsPlugin.java`

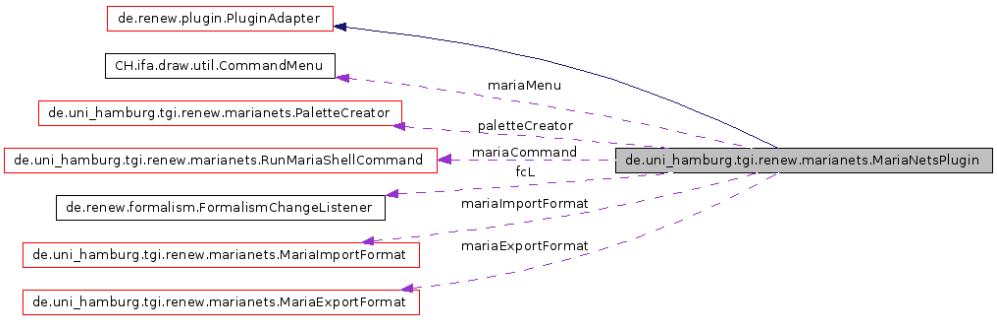

Abbildung 7: Kollaborationsgraph für die Klasse `MariaNetsPlugin`

Die bereits in Abschnitt 2.2 (S.6) beschriebene `init` Methode registriert den Compiler `MariaCompiler`, fügt Import und Export Formate (`MariaImportFormat` bzw. `MariaExportFormat`) sowie den Menüpunkt zum Start der Maria Konsole (Klasse `MariaConsole`) hinzu. Außerdem wird ein `FormalismChangeListener` (innere Klasse) registriert, der beim Wechsel auf den Maria-Formalismus eine Schaltfläche für den Query Node sichtbar macht.

Die `cleanup` Methode entfernt alle in `init` hinzugefügten Elemente wieder.

Das Kollaborationsdiagramm[11] in Abbildung 7 dient als Ausgangspunkt für die Beschreibung der weiteren Klassen, nach den Hauptfunktionen *Parsing*, *Export* und *Kommunikation* und nach Quelldateien gruppiert.

Zum Teil entspricht die Arbeitsteilung und Benennung von Klassen der von anderen Plugins, um Programmierern, die sich mit anderen Plugins auskennen, eine schnellere Einarbeitung in das `MariaNets` Plugin zu ermöglichen.

7.2.2 Parsing

wird von mehreren Dateien implementiert, deren Kernstück die in `MariaParser.jj` beschriebene Grammatik bildet.

[11] Der Kollaborationsgraph in Abbildung 7 wurde mit dem Programm DOXYGEN (`http://www.doxygen.org/index.html`) automatisch aus den Quelltextdateien erstellt. Die durchgezogenenen Linien bedeuten, dass die vorliegende Klasse von der Klasse erbt, auf die der Pfeil zeigt (in diesem Fall `de.renew.plugin.PluginAdapter`) und die gestrichelten Linien verweisen auf Aggregation, d. h. Objekte der Klasse `MariaNetsPlugin` verwenden Objekte der angezeigten Klassen als Felder. Wenn ein Klassenname rot eingerahmt ist, bestehen weitere Beziehungen dieser Klasse zu anderen Klassen, die in der Grafik nicht angezeigt werden. `RunMariaShellCommand` beispielsweise erbt von einer weiteren Klasse und implementiert ein Interface.

`MariaParser.jj` In den RENEW-Quellen war im Quellbaum des Gui-Plugin (`MariaParser.jj`) von Olaf Kummer bereits ein in JAVACC implementierter Parser für MARIA-Anschriften vorhanden, der als Grundgerüst tauglich war, aber offenbar für eine wesentlich ältere MARIA Version geschrieben wurde und daher vollständig für die aktuelle MARIA Version überarbeitet werden musste.

Ausgehend von der alten Version wurden zunächst die Tokendefinitionen und dann die BNF Produktionen aktualisiert[12], die zunächst noch alle den Rückgabetyp *`void`* behielten.

Um die Aufspaltung von Anschriften besser zu unterstützen, wurden noch einige BNF Produktionen mit dem Rückgabewert `bool` hinzugefügt, die einigermaßen sinnvoll raten, ob eine Stellenanschrift ein Typ, ein Constraint oder eine Markierung beschreibt, bzw. ob eine Transitionsanschrift am Anfang, am Ende oder (im Falle von Funktionen) in der Mitte der Transition stehen muss. Da dieses Raten bei Stellen nicht immer das richtige Ergebnis erzeugt, ist es in Zweifelsfällen ratsam, Typ, Constraint und Markierung in eine einzige Anschrift zu schreiben. Aus dem gleichen Grund wird auch beim Import von Maria Netzen aus diesen drei Anschriften eine einzige erzeugt.

Danach wurde der Parser sehr umfangreich überarbeitet, so dass sich mit der BNF Produktion `drawing()` ein Objekt vom Typ `CPNDrawing` erzeugen lässt, was auch beim Import genutzt wird. Die Syntax wurde nicht verändert, aber in allen relevanten BNF Produktionen wurden Java-Aktionen hinzugefügt, die Elemente der RENEW-Zeichnung (wie Figuren, Verbindungen oder Zeichenketten) erzeugen und verbinden. Alle Beispiele aus den MARIA Quellen, die keine verschmolzenen Stellen oder Subnetze verwenden, konnten damit erfolgreich importiert werden (also in der Version 1.3.5 von MARIA alle Beispielnetze außer `swn-m.pn` und `modular.pn`), wobei das Netz `swn.pn` nicht immer nach dem Export funktioniert, wie in Abschnitt 6.5 (S.24) beschrieben. Es ist auch auf Grund der dort beschriebenen Probleme sehr gut möglich, dass auch andere gültige nicht-modulare MARIA Netze nach dem Import oder Export in RENEW nicht mehr korrekt funktionieren, so dass entweder die Anschriften am grafischen Netz in RENEW, oder die Ausgabe nach dem Export angepasst werden müsste. In der Beschreibung der Klasse `MariaCreator` in Abschnitt 7.2.3 (S.31) wird auf mögliche Lösungen für diese Problematik eingegangen.

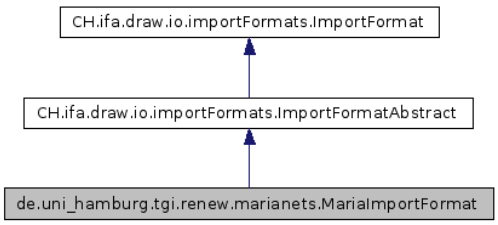

Abbildung 8: Kollaborationsgraph für die Klasse `MariaImportFormat`

`MariaImportFormat.java` In `MariaNetsPlugin` wird ein Objekt dieser Klasse erzeugt und als Import Format registriert. Die Methode `importFiles` ruft die Methode `parse` in der Klasse `MariaFormat` auf und sorgt danach bei jedem der im-

[12] Für die Terminologie siehe [CollabNet, 2006a]

portierten Netze lediglich dafür, dass der Name richtig gesetzt wird, der der Klasse `MariaFormat` nicht bekannt ist.

Das Zusammenspiel der Klassen `MariaFormat`, `MariaImportFormat`, `MariaExportFormat` und `MariaFileFilter` wurde den Import- und Exportfiltern für PNML Netze entlehnt. Der Hauptgrund für die Aufteilung von `MariaImportFormat` und `MariaExportFormat` besteht darin, dass für das Registrieren der Filter in `MariaNetsPlugin` Objekte vom jeweils passenden Typ benötigt werden.

`MariaFormat.java` Diese Klasse stellt die Methode `write` zur Verfügung, die von der Methode `export` in `MariaExportFormat` benötigt wird und lediglich an die Methode `write` in `MariaCreator` delegiert.

Ebenso delegiert die Methode `parse` lediglich an die BNF Produktion `drawing` in der von JAVACC generierten `MariaParser` Klasse.

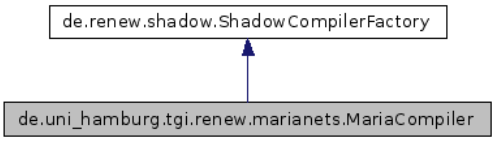

Abbildung 9: Kollaborationsgraph für die Klasse `MariaCompiler`

`MariaCompiler.java` Diese Klasse realisiert das Design Pattern „Fabrik“. Mit der Methode `createCompiler` lassen sich Objekte vom Typ `SingleMariaNetCompiler` erzeugen. Diese Klasse stammt von Olaf Kummer und konnte (bis auf die Anpassung des Paketnamens) unverändert verwendet werden.

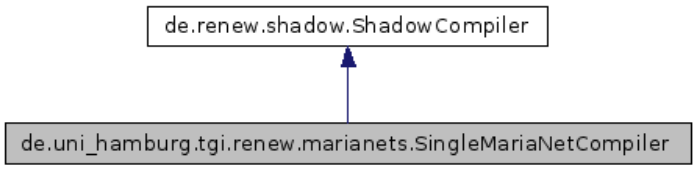

Abbildung 10: Kollaborationsgraph für die Klasse `SingleMariaNetCompiler`

`SingleMariaNetCompiler.java` Die Methoden dieser ursprünglich von Olaf Kummer geschriebenen Klasse rufen BNF Produktionen des Parsers auf, um die Syntax Überprüfung direkt nach dem Erstellen von Netzanschriften zu ermöglichen. Die Methode `CheckDeclarationNode` wurde von mir um die Syntax Prüfung des Query Nodes erweitert.

`MariaFileFilter.java` Diese einfache Klasse von Lawrence Cabac und mir dient zum Filtern von Dateien mit der Erweiterung `.pn`. Die Beschreibung wurde von mir angepasst.

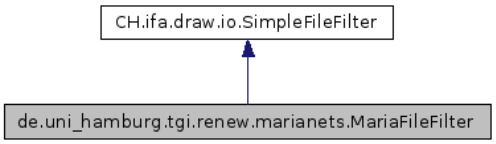

Abbildung 11: Kollaborationsgraph für die Klasse `MariaFileFilter`

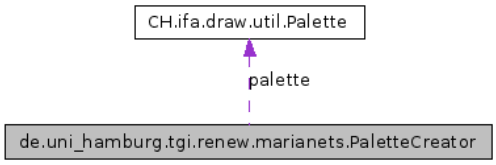

Abbildung 12: Kollaborationsgraph für die Klasse `PaletteCreator`

`PaletteCreator.java` In diese Klasse wurde der Code zum Anlegen der Palette für Maria Netze ausgelagert. Die Palette enthält zur Zeit nur das Query Tool (mit einem „q“ als Icon), aber diese Abstraktion wurde dennoch aus verschiedenen anderen existierenden Plugins übernommen.

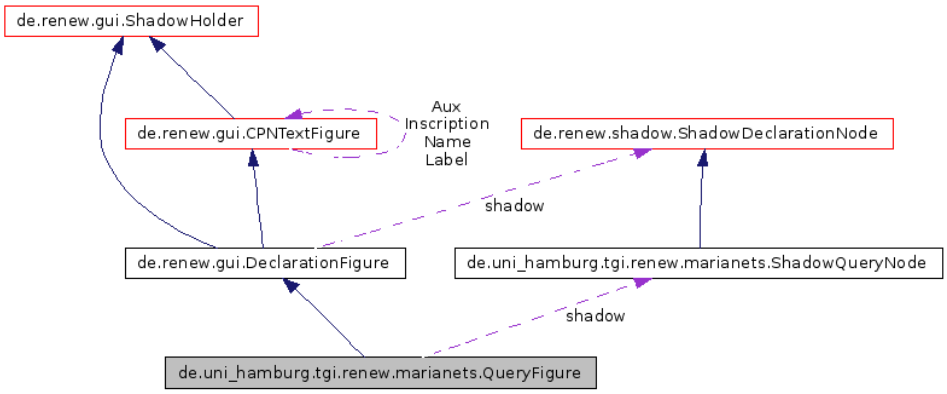

Abbildung 13: Kollaborationsgraph für die Klasse `QueryFigure`

`QueryFigure.java` Diese Klasse erbt `DeclarationFigure`, weil dadurch das in `SingleMariaNetCompiler` implementierte Interface `ShadowCompiler` unverändert übernommen werden kann und einfach in der Methode `CheckDeclarationNode` die Anschrift des Query Node geprüft werden kann. Die Anschrift der Query Figure ist in *italic* gehalten und sie hat als Objekt vom Typ `DeclarationFigure` den Typ `AUX`. Beim Schreiben der Figur in einen Stream wird die Integer-Zahl `QUERY` zusätzlich in den Stream geschrieben, welche beim Lesen wieder erwartet wird.

Aktuell wird beim Bearbeiten der Anschrift der Query Figure in der Werkzeugleiste die Schaltfläche des Declaration-Tools selektiert. Das wurde in der aktuellen Version in Kauf genommen, weil soweit ich sehen konnte zum Beheben des Problems ein Eingriff in die entsprechende Klasse im `Gui` Plugin nötig gewesen wäre, aber diese falsche Anzeige keine Beeinträchtigung der Funktionalität des Query Node zur Folge hat.

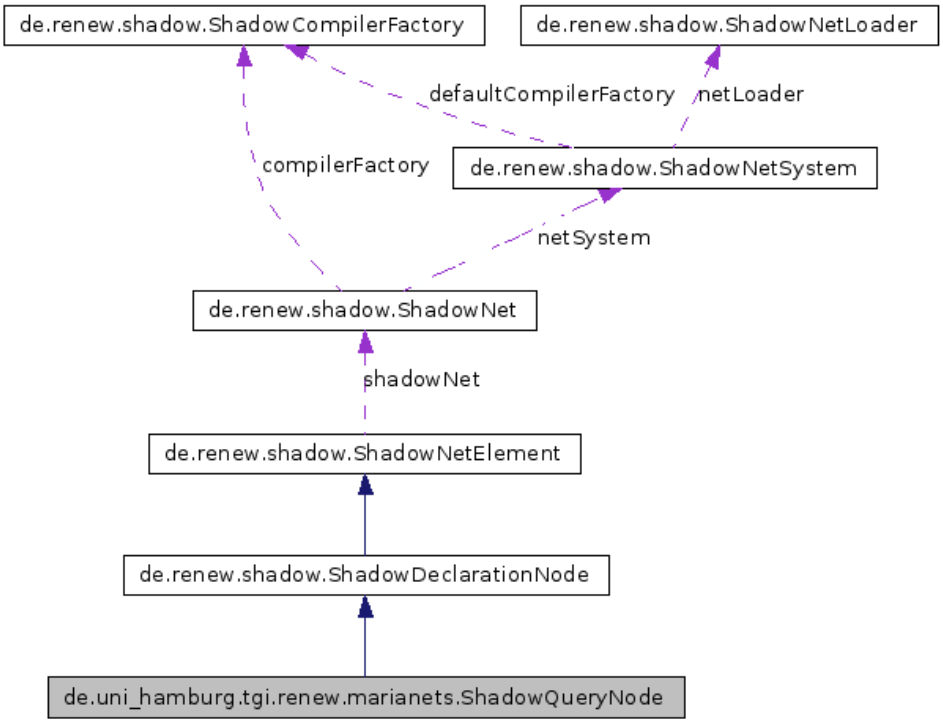

Abbildung 14: Kollaborationsgraph für die Klasse `ShadowQueryNode`

`ShadowQueryNode.java` Obwohl MARIA Netze aktuell nicht simuliert werden, dient der `ShadowQueryNode` zur Unterscheidung von seiner Superklasse (siehe Abbildung 14) bei der Syntaxprüfung und beim Export.

`MariaQueries.java` Hier werden verschiedene statische Methoden gekapselt, die von verschiedenen Klassen verwendet werden, die mit Query Nodes und ihren Inschriften zu tun haben (`MariaConsole`, `MariaCreator`, `RunMariaShellCommand` und `SingleMariaNetCompiler`).

7.2.3 Export

wird im wesentlichen durch `MariaCreator` erledigt.

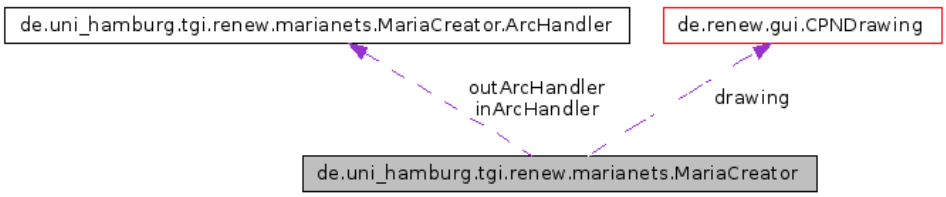

Abbildung 15: Kollaborationsgraph für die Klasse `MariaCreator`

`MariaCreator.java` Hier wird in der Methode `write` eine Zeichnung mit MARIA Anschriften in einen Ausgabestrom (eine geöffnete Datei) geschrieben.

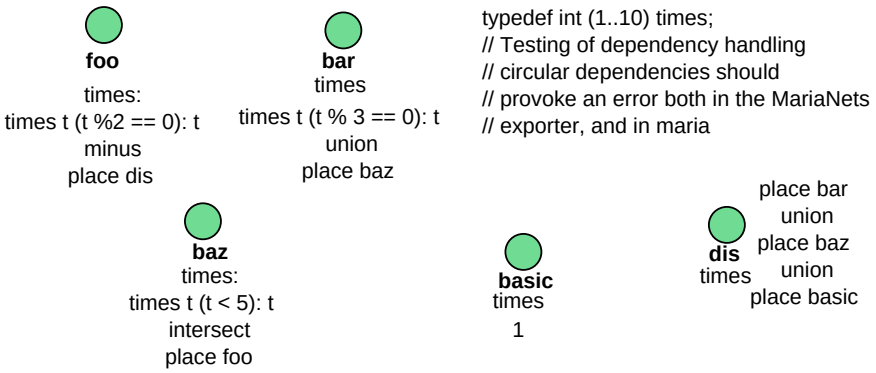

Abbildung 16: Das Netz „circular“

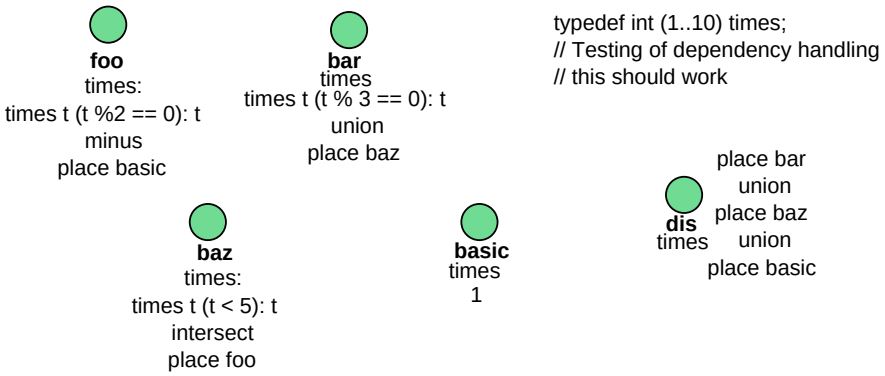

Abbildung 17: Das Netz „noncircular“

Die Klasse stammt ursprünglich ebenfalls von Olaf Kummer, und in der alten Version wurde bereits die Trennung von Anschriften in verschiedene syntaktische Bestandteile ermöglicht. Diese Funktionalität wurde von mir erweitert. Außerdem werden die Inschriften der Query Nodes als Kommentare am Ende der Datei angehängt.

Abhängigkeiten zwischen verschiedenen Inschriften waren allerdings bisher vollkommen unberücksichtigt geblieben. Diese treten in MARIA Netzen sehr häufig auf, insbesondere bei Stellen, aber auch in Funktionen (wie bereits in Abschnitt 6.5 (S.24) erläutert) und sogar in Kanteninschriften sind Abhängigkeiten erlaubt.

Dabei handelt es sich in allen Fällen um Referenzen auf Bezeichner, die in textuell vorhergehenden Netzelementen gebunden und/oder erzeugt wurden.

Es gibt in Renew Netzen keine leicht verwendbare und klar verständliche Möglichkeit eine Reihenfolge der Anschriften zu definieren, also muss eine Lösung darauf beruhen, dass die Abhängigkeiten zwischen den Inschriften und damit die richtige Reihenfolge der Netzelemente erkannt werden..

Das Beispielnetz in Abbildung 16 enthält zirkuläre Referenzen in den Anfangsmarkierungen der Stellen `foo`, `bar`, `baz` und `dis`. Im Falle von zirkulären Abhängigkeiten gibt es keine richtige Reihenfolge, also muss ein Algorithmus zur Erkennung

von Abhängigkeiten solche Zyklen erkennen und eine Fehlermeldung ausgeben, da das exportierte Netz in MARIA ohnehin nicht funktionieren könnte. Ein viel wichtigeres Problem besteht darin, dass ein Algorithmus zur Erkennung der Abhängigkeiten auch im Falle von zyklischen Abhängigkeiten kontrolliert terminieren muss.

Das Beispielnetz in Abbildung 17 links gegenüberliegend soll korrekt exportiert werden und zwar ist die richtige Reihenfolge der Stellen in diesem Fall:

1. `basic`
2. `foo`
3. `baz`
4. `bar`
5. `dis`

Algorithmus zum Ordnen von Stellen nach ihren Abhängigkeiten: Die Funktion `reorderPlaces` gibt den als Parameter erhaltenen Vektor des Typs `ShadowPlace` nach Abhängigkeiten sortiert zurück. Der Algorithmus geht wie folgt vor:

1. Zunächst werden für jeden Platz im Vektor die in Frage kommenden Anschriften auf Referenzen untersucht. Wird eine Referenz auf eine andere Stelle gefunden, werden die Stellen in einen gerichteten Graph (`references`) eingetragen.

2. Dann werden alle Knoten des Graphen in eine Queue (`placeQueue`) eingetragen.

3. Dann werden alle Knoten in der Queue nacheinander durchlaufen:

   (a) Knoten ohne ausgehende Kanten (also Stellen, die keine anderen referenzieren), werden in den Hash `placesByTier` unter dem Schlüssel „0“ eingetragen und mit „0“ markiert.

   (b) Plätze, die nur mit bereits markierten Plätzen direkt durch ausgehende Kanten verbunden sind, werden mit dem Maximum der gefundenen Markierungen + 1 markiert und auch unter dieser Zahl im Hash eingetragen.

   (c) Plätze, die durch ausgehende Kanten mit noch unmarkierten Plätzen verbunden sind, werden wieder in die Queue zurückgestellt.

   Die Queue wird so lange durchlaufen, bis sie entweder leer ist, oder bis sie einmal komplett durchlaufen wurde, ohne ihre Länge zu verändern (d. h. dass beim Durchlauf kein Knoten markiert werden konnte). Im letzten Fall wird eine Ausnahme geworfen und eine Fehlermeldung an den Benutzer gegeben.

Der Algorithmus ist momentan sehr speziell auf Stellen ausgerichtet und vermutlich auch nicht die effizienteste Lösung des Problems.

Da Abhängigkeiten zwischen Stellenanschriften sehr häufig auftreten und erst später aufgefallen ist, dass auch an anderen Netzelementen problematische Abhängigkeiten auftreten können, werden in der aktuellen Version von `MariaCreator` nur Abhängigkeiten zwischen Stellenanschriften aufgelöst, aber ich glaube, dass sich der Algorithmus auch auf Transitionen, Funktionen und Kanten verallgemeinern lässt, allerdings sind Bindungen an Kanten etwas schwerer zu erkennen als Referenzen auf Stellen, so dass eventuell der Parser zur Unterstützung angepasst werden muss.

Die übrigen Methoden in `MariaCreator` sorgen dafür, dass die Netzelemente und Anschriften in der Reihenfolge in den Ausgabestrom geschrieben werden, die in den meisten Fällen zu einem funktionierenden MARIA Netz führt.

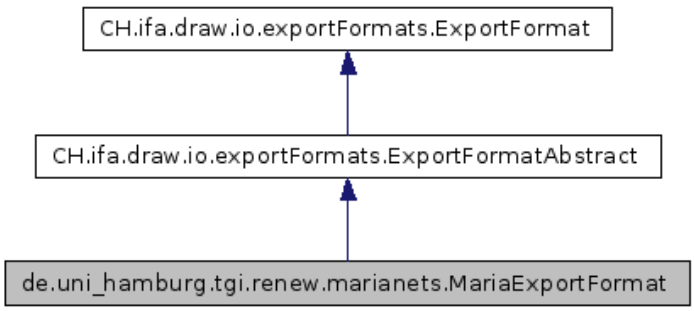

Abbildung 18: Kollaborationsgraph für die Klasse `MariaExportFormat`

`MariaExportFormat.java` Diese Klasse delegiert an die Methode `write` in der Klasse `MariaFormat`.

7.2.4 Kommunikation

und ihre Darstellung benötigte etwas Abstraktion, da die Kommunikation mit einem externen Prozess bisher in keinem RENEW-Modul benötigt wurde. Für die Darstellung konnte ich mich zur Anregung auf das `GuiPrompt` Plugin stützen.

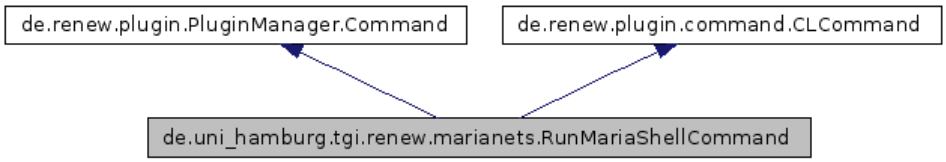

Abbildung 19: Kollaborationsgraph für die Klasse `RunMariaShellCommand`

`RunMariaShellCommand.java` Diese Klasse erbt von `Command` und implementiert `CLCommand`. Dadurch kann die Maria Konsole sowohl an der RENEW Kommandozeile (mit dem Befehl „`maria`“ ohne Parameter) als auch durch einen Menüpunkt aufgerufen werden (in `Plugins` - `Maria` - `Maria Console`).

In beiden Fällen wird zunächst die aktuelle Zeichnung mittels `MariaExportFormat` exportiert, die Anfragen aus dem Query Node extrahiert, und dann die Maria Konsole gestartet.

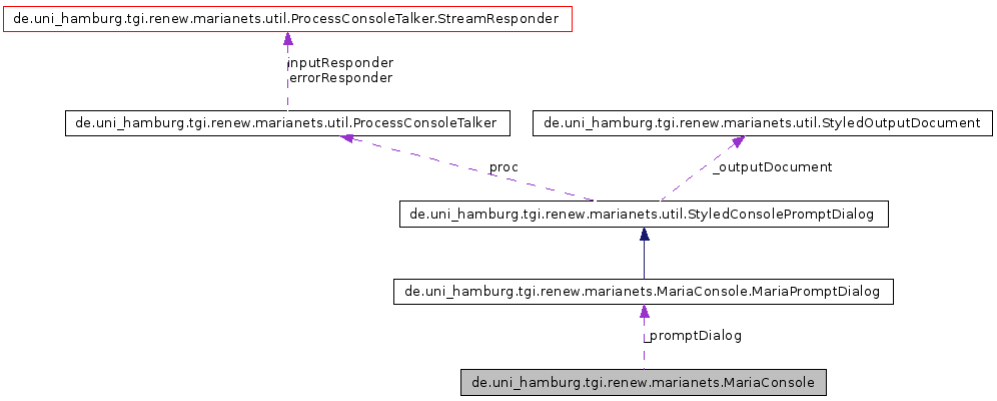

Abbildung 20: Kollaborationsgraph für die Klasse `MariaConsole`

`MariaConsole.java` In der Klasse `MariaConsole` wird sämtliche Arbeit von der inneren Klasse `MariaPromptDialog` erledigt. Hier werden Gui Elemente erzeugt und über den Konstruktor der Superklasse `StyledConsolePromptDialog` das Programm gestartet und die initialen Anfragen aus dem Query Node abgesetzt. Es werden nur die Methoden aus `StyledConsolePromptDialog` überschrieben, die speziell an MARIA angepasst werden müssen. Dadurch soll die Superklasse allgemein verwendbar bleiben und könnte z. B. für andere Programme oder den `GuiPrompt` angepasst werden.

Insbesondere wird der GUI ein „`kill`“ Knopf hinzugefügt, mit dessen Hilfe eine lange Berechnung in MARIA abgebrochen werden kann (indem einfach der Prozess beendet wird). Gegenüber dem Schließen des Konsolenfensters hat dies den Vorteil, dass die bisher gesammelte Ausgabe des zuletzt abgeschickten Kommandos noch angezeigt wird.

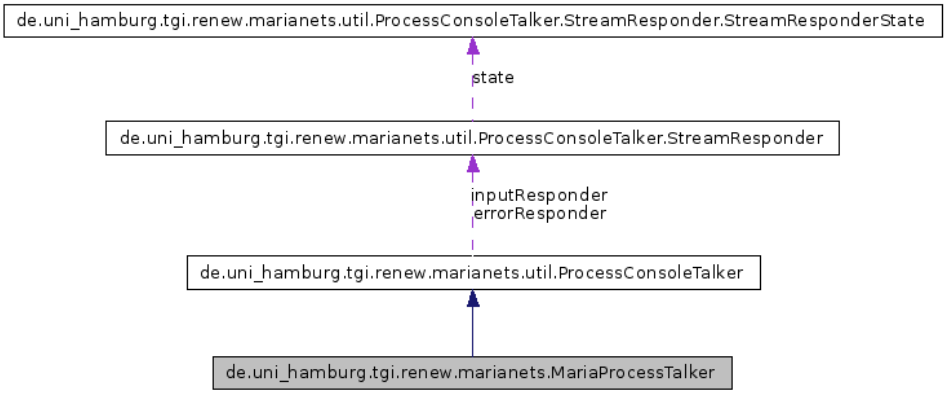

Abbildung 21: Kollaborationsgraph für die Klasse `MariaProcessTalker`

`MariaProcessTalker.java` spezialisiert die Klasse `ProcessConsoleTalker` für die Anforderungen des `maria` Prozesses.

Das Kommando und seine Parameter werden aus der Plugin Property „`marianets.mariaCommand`“ gelesen. Die Voreinstellung hierfür ist „`maria`“. Mit der Einstellung

„maria -p lbt“ könnte zwar darauf verzichtet werden im Query Node mit translator lbt den Übersetzer für LTL Formeln anzugeben, aber das MariaNets Plugin würde mit dieser Einstellung nicht mit der Windows Version von MARIA funktionieren, weil dort der Schalter „-p“ standardmäßig nicht existiert und zum Programmabbruch führt, da der Übersetzer für LTL Formeln standardmäßig in das Programm eingebaut ist. Also sollte aus Portabilitätsgründen der Übersetzer im Query Node angegeben werden.

Sowohl maria als auch (unter Unix) lbt müssen sich im Pfad befinden. Alternativ kann der Pfad für maria auch in der Property „marianets.mariaCommand“ explizit angegeben werden, der Übersetzer lbt kann aber nur im Query Node angegeben werden und es gibt z. Z. keine Möglichkeit das Visualisierungsskript maria-vis zu spezifizieren, wenn es nicht im Pfad liegt.

Es gibt ein weiteres Portabilitätsproblem: Die Windows Version von MARIA schreibt den Prompt und alle anderen Ausgaben in den „Error Stream“, der mit der Methode getErrorStream der Klasse java.lang.Process erhalten wird, während die Unix Versionen den Prompt in den „Input Stream“, alle anderen Ausgaben jedoch in den „Error Stream“ schreiben[13]. Da die Klasse ProcessConsoleTalker den Prompt finden muss, um das Ende der Antwort sicher zu erkennen, muss in der hier beschriebenen Klasse dieses Verhalten berücksichtigt werden.

Auf einem Windows System wird darum davon ausgegangen, dass der Prompt im „Error Stream“ erscheint, was auf anderen Systemen mit der Plugin Property „marianets.mainPromptOnStderr“ erzwungen werden kann. Wird diese Property falsch gesetzt, führt dies zwangsläufig dazu, dass die Kommunikation bereits im Setup hängen bleibt, was in dieser frühen Phase dazu führt, das die RENEW Gui ebenfalls nicht mehr benutzbar ist.

Weitere Anpassungen betreffen das Senden von unvollständigen Kommandos, das Entfernen vom Benutzerprompt „#“[14] aus dem Fehlerstrom und das erzwungene Beenden des maria Prozesses.

ProcessConsoleTalker.java ProcessConsoleTalker kapselt die Kommunikation mit einem Prozess. Es können Prompts für beide Antwortströme (in Java: „Error“ und „Input Stream“) sowie eine lange und eine kurze Wartezeit eingestellt werden. Das Lesen der Antwort wird an zwei Threads StreamResponder delegiert, welche wiederum einen einfachen endlichen Automaten zur Verwaltung ihres Zustandes verwenden.

Das Verhalten des Objekts kann zur Laufzeit mit einigen Getter- und Setter-Methoden abgefragt und verändert werden, letzteres wird von MariaProcessTalker gebraucht, um das Objekt für unvollständige Anfragen zu konfigurieren (in dem Fall ändert sich der Prompt des maria Prozesses signifikant).

Bei jeder mit der Methode sendRequest (und ihren Delegaten) gestellten Anfrage

[13] Es muss angemerkt werden, dass diese Verwendung des Fehlerstroms durch MARIA nicht den, zumindest unter Unix, üblichen Konventionen entspricht. Darüber hinaus ist diese Inkonsistenz zwischen Unix und Windows mit Sicherheit ein Bug, der bisher nicht bemerkt wurde, da niemand vorher ein Frontend für MARIA geschrieben hat. Da MARIA aber nicht mehr entwickelt wird und alle unter Windows getesteten Versionen (1.3.4 und 1.3.5) dieses Verhalten aufweisen, ist es am besten, wenn sich das MariaNets Plugin daran anpasst.

[14] Dieser wird in den Unix Versionen benötigt, um das Ende der Ausgabe sicher zu erkennen, da die Ausgabe immer auf dem Fehlerstrom, der Hauptprompt aber auf dem Eingabestrom erscheint.

wird der Automat in den Zustand „committed“ versetzt, in dem auf einen Prompt gewartet wird. Wird ein Prompt im Antwortstrom gefunden, wird dies durch den Zustand „prompt_seen“ signalisiert, in welchem Fall die Antwort abgespeichert wird und von den Methoden getLastInput und getLastError (bzw. delegiert durch andere Methoden) abgeholt werden kann. Danach folgt automatisch der Zustand „collecting“ in welchem die Daten im Antwortstrom eingesammelt werden und auf weitere Anfragen gewartet wird. Der Zustand „done“ kann durch die Methode finish oder das Ende eines der Antwortströme erreicht werden und führt zum Beenden des jeweiligen StreamResponder Threads.

Da sowohl der Haupt-Thread als auch der StreamResponder Thread selbst auf den Zustandsautomaten zugreifen muss, ist dieser durch einen Monitor abgesichert.

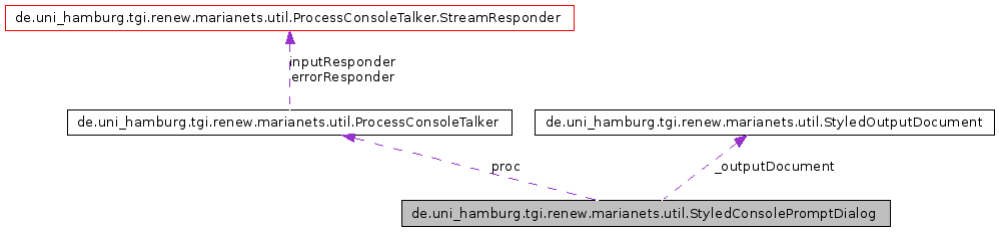

Abbildung 22: Kollaborationsgraph für die Klasse StyledConsolePromptDialog

StyledConsolePromptDialog.java bietet einen Dialog an, mit dem über ProcessConsoleTalker und StyledOutputDocument interaktiv mit einem Prozess kommuniziert werden kann. Dabei werden im Ausgabedokument die Benutzereingaben in **fett** gesetzt, der „Input Stream“ in blau und der „Error Stream“ in rot. Möglicherweise können diese Einstellungen später über Plugin Properties konfigurierbar gemacht werden.

Diese Einfärbung wurde vorgenommen, damit in der Ausgabe eine leichtere Unterscheidung zwischen Benutzereingaben und Ausgaben des Programms getroffen werden kann. Die Methode execShowBusyAndShowReply bildet das Kernstück der Funktionalität. Sie schickt eine Anfrage ab und sorgt dafür, dass sie so bald als möglich angezeigt wird und die Gui-Elemente aktualisiert werden, ohne dass die übrigen Programmteile beeinträchtigt werden.

Dazu wird auf einen zusätzlichen Thread (siehe [Sun Microsystems, 2006]) zurückgegriffen.

Da die Antworten in einem separaten Thread abgearbeitet werden, muss darauf geachtet werden, dass die Abfolge von Benutzereingaben und Ausgaben des Programms weiterhin natürlich aussieht. Zur Synchronisation mit dem Thread wird deshalb ein Monitor Objekt verwendet.

StyledOutputDocument.java Ursprünglich inspiriert durch die Klasse de.renew.gui.ResponseDocument sollte diese Klasse nur eine Abstraktion von multiplen Strömen mit unterschiedlichen Stilen (Textauszeichnung, Farben o.ä.) zur Verfügung stellen, über die beispielsweise Syntax-Highlighting ermöglicht werden kann. Da das Multithreading in ResponseDocument jedoch Probleme verursachte, insbesondere

wenn es auf mehrere Threads und Ströme erweitert wurde und derartige Probleme bereits durch `ProcessConsoleTalker` gelöst werden, wurde die Thread Funktionalität (und damit die Möglichkeit des Lesens eines Eingabestroms) aus der Klasse entfernt, womit nur noch eine sehr einfache Erweiterung der `StyledDocument` Klasse aus der `javax.swing` Hierarchie übrig geblieben ist.

Jeder der Ströme wird mit Attributen (`javax.swing.text.AttributeSet`) versehen, die Standardanzahl von Strömen ist `3`, womit z. B. Ein-, Ausgabe- und Fehlerstrom unterschiedlich ausgezeichnet werden können, aber die Anzahl der Ströme und die Attribute können im Konstruktor frei ausgewählt werden.

Es wäre denkbar, `ProcessConsoleTalker` weiter zu abstrahieren und insbesondere die `StreamResponder` Klasse in eine eigene Quelldatei auszulagern, so dass aus ihr und `StyledOutputDocument` ein Aggregat gebildet werden kann, welches die Funktionalität von `de.renew.gui.ResponseDocument` mit Textauszeichnung zur Verfügung stellt, falls spätere Plugins solche Funktionalität benötigen sollten (oder im `GuiPrompt` Plugin zwischen Ein- und Ausgaben unterschieden werden soll).

8 Anwendung

Wenn das `MariaNets` Plugin in das `plugins` Verzeichnis der RENEW Installation (oder ein weiteres Plugin Verzeichnis, siehe [Kummer u. a., 2006b, S.22]) kopiert wurde,[15] steht im Menü `File` in den Untermenüs `Import` und `Export` das MARIA Dateiformat zur Verfügung, außerdem kann im Untermenü `Simulation - Formalisms` der MARIA Formalismus gewählt werden.

In Abbildung 23 auf der nächsten Seite ist das Netz aus Listing 2 (S.14) als RENEW-Netz zu sehen.

Oben links ist ein Deklarationsknoten mit frühen Deklarationen, darunter ein weiterer Deklarationsknoten mit einer späten Deklaration *`deadlock true`*.

Namen von Stellen und Transitionen sind voreingestellt **fett**, die Anschrift des Query Node (rechts oben) *italic* und alle anderen Anschriften in normaler Schrift.

Allerdings sind dies nur die voreingestellten Schriftauszeichnungen, zur Hervorhebung können die Auszeichnungen aller Anschriften (wie sonst auch bei RENEW) beliebig verändert werden. Der Fairness-Constraint bei der Transition **`left`** wurde in **fetter** Schrift hervorgehoben.

Sowohl Stellen als auch Transitionen *müssen* Namen haben, Stellen brauchen darüber hinaus mindestens eine Typdeklaration und Kanten brauchen eine unifizierbare Inschrift, damit das Netz in MARIA funktioniert. Das `MariaNets` Plugin testet momentan keine dieser Bedingungen vor dem Export und schlägt auch keine für MARIA sinnvollen Voreinstellungen vor.

Die Stellen und Transitionen tragen Anschriften wie im Listing, wobei in diesem Beispiel die Constraints, Typdeklarationen und initialen Markierungen aufgespalten wurden, was in diesem Fall zu eindeutigen Ergebnissen führt. Sollte es beim Export

[15] Bei RENEW Version 2.1 ist zusätzlich ein modifiziertes `ch` Plugin erforderlich, da die Klasse `StorableInput` angepasst werden musste. Andernfalls können zwar MARIA Netze mit Query Node erstellt und abgespeichert werden, der Query Node ist aber nach dem Laden verschwunden. Vor RENEW 2.1 wird das `MariaNets` Plugin vermutlich gar nicht funktionieren. Zu allen RENEW Versionen nach 2.1 (also auch der CVS Version) sollte das Plugin vollständig kompatibel sein.

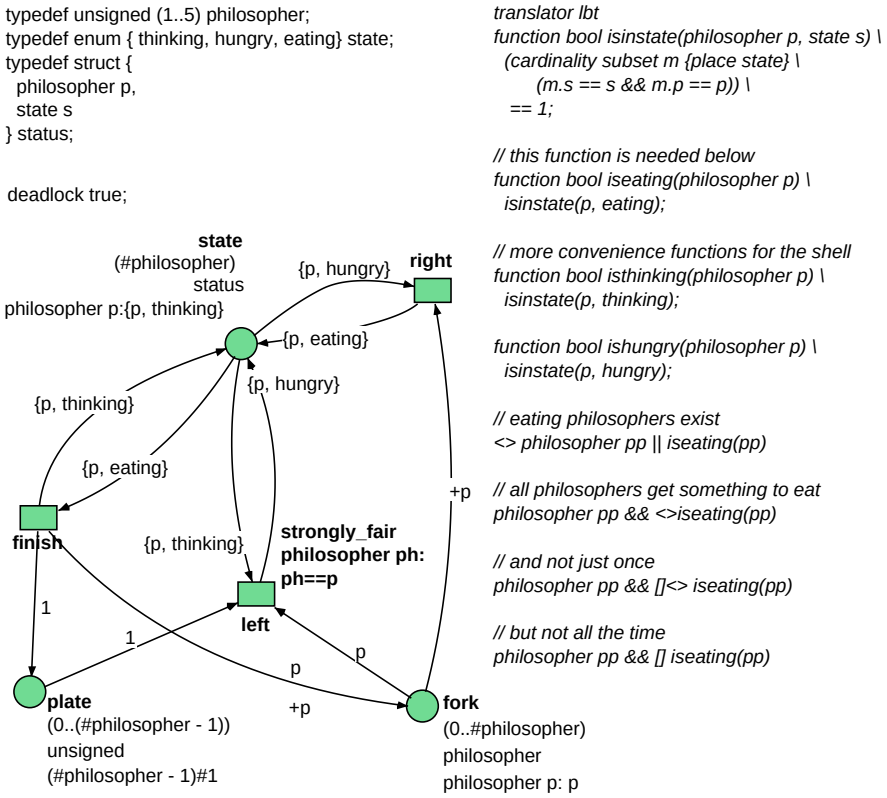

Abbildung 23: Das Netz „dining-fair+safe“

Probleme geben, könnten z.B. Constraint, Typdeklaration und Markierung der Stelle **`state`** wie folgt zu einer Anschrift zusammengefasst werden: `(#philosopher) status: philosopher p:{p, thinking}` wobei hier der Doppelpunkt hinter der Typdeklaration folgen *muss* im Unterschied zur aufgespaltenen Anschrift, wo er *nicht* stehen *darf*.

Stellenanschriften können aufgespalten werden, sofern die initiale Markierung eindeutig als solche zu erkennen ist, und nicht für einen Namen gehalten werden kann, der dann *immer* als Typdeklaration verstanden wird. *Erlaubt* sind also z. B. „`{p, eating }`“, „`1`“, „`4#p`“, „*`is`* `state eating`“, *nicht erlaubt* sind z. B. „`eating`“, „`p`“. Außerdem würde eine Markierung, die (aus welchem Grund auch immer) unnötigerweise komplett in runde Klammern eingeschlossen ist, für einen Constraint gehalten.

Wenn im Menü `Simulation - Formalisms` der MARIA Formalismus gewählt wird, wird nach dem Editieren der Anschriften die Syntaxprüfung durchgeführt und bei fehlerhaften Anschriften ein Fehler gemeldet. Die vom `MariaParser` geworfenen `ParseExceptions` sind allerdings z. Z. weitaus ausführlicher als die des Java Parsers, was am semantischen Lookahead liegt (siehe [CollabNet, 2006a] und [CollabNet, 2006b]).

Mit einem weiteren Menüpunkt `Maria Console` im Untermenü „`Plugins - Maria`“ wird die MARIA Konsole gestartet und die Anfragen im Query Node werden abgeschickt. Nachdem die letzte Anfrage im Anfrageknoten (`philosopher pp && [] iseating(pp)`) abgearbeitet ist, kann der Benutzer eigene Anfragen in der Eingabezeile absetzen, wie z. B. *`dumpgraph`* (um gleich darauf die Funktionalität des „`kill`“ Buttons zu testen, weil diese Anfrage ein wenig dauert).

Vor manchen Befehlen erlaubt MARIA auch die Eingabe *`visual`*, hierfür muss das Skript `maria-vis` im Pfad liegen. Unter Unix Derivaten wird die Ausgabe dann in einem interaktiven Browser angezeigt (über den `MariaConsole` keine Kontrolle ausübt), sofern eine GRAPHVIZ-Installation zur Verfügung steht (siehe auch [Mäkelä u. a., 2005]).

9 Zusammenfassung und Ausblick

Nachdem diese Arbeit die Petrinetz Tools RENEW und MARIA vorgestellt und ihre Netzformalismen miteinander verglichen hat, wurden mehrere Alternativen ihrer Integration diskutiert. Die daraus resultierende Implementation bietet die Möglichkeit, MARIA Netze grafisch darzustellen und in grafischer Form zu bearbeiten und kann darüber hinaus auch Anfragen zusammen mit dem Modell abspeichern, was in MARIA allein so nicht möglich ist. Darüber hinaus bietet die Maria Konsole durch die unterschiedliche Auszeichnung von Ein- und Ausgaben eine bessere Übersicht als die `maria` Shell.

Aus dem aktuellen Zustand der Implementation ergeben sich aber auch Probleme, die vorher nicht ersichtlich waren, sowie Ideen für die Verbesserung und Erweiterung des Prototypen.

Zu den Problemen gehören die diskutierten Abhängigkeiten zwischen Anschriften. Die gefundene Lösung für Stellenmarkierungen könnte auf alle Inschriften erweitert werden, damit das in RENEW erstellte MARIA Netz auch in jedem Fall nach dem Export funktioniert.

Außerdem ist es noch unbefriedigend, dass modulare Netze nicht importiert oder erstellt werden können. Der aufgezeigte Denkansatz könnte genauer spezifiziert und implementiert werden, damit ein sehr zentrales MARIA Feature auch von RENEW aus verfügbar wird.

Denkbare Erweiterungen des Plugins wären die Einbindung in den Simulator und eine grafische Darstellung der Ergebnisse des Model Checkers. Es wäre zum Beispiel denkbar, die Gegenbeispiele grafisch als Petrinetz Prozesse darzustellen, womit einerseits das Problem der uneindeutigen grafischen Darstellung in `maria-vis` gelöst, als auch ein neues Werkzeug zur Netzexploration in RENEW geschaffen würde. Eine kleinere Vervollständigung der Implementation wäre durch das Verfolgen von `#include` Direktiven gegeben.

Nachdem das `MariaNets` Plugin auf diese Weise verbessert und erweitert wurde, werden sich zweifellos weitere Probleme und Ideen ergeben und nicht zuletzt Werkzeuge entstanden sein, die das große Projekt einer direkten Einbindung von effizienten Model Checking Verfahren für Referenznetze etwas erleichtern sollten.

Literatur

[Aalto 2004] AALTO, Annikka: Automatic Translation of SDL into High Level Petri Nets / Helsinki University of Technology, Laboratory for Theoretical Computer Science. Espoo, Finland, November 2004 (B21). – Technical Report. – 66 S. – URL http://www.tcs.hut.fi/Publications/B21.shtml. URL besucht am 07. August 2006. – ISBN 951-22-7404-3 19

[Christensen und Hansen 1994] CHRISTENSEN, Søren ; HANSEN, Niels D.: Coloured Petri Nets Extended with Channels for Synchronous Communication. In: *Application and Theory of Petri Nets*, URL http://www.daimi.au.dk/PB/390/PB-390.pdf, 1994, S. 159–178. – URL besucht am 05. August 2006 19

[Christensen und Petrucci 1995] CHRISTENSEN, Søren ; PETRUCCI, Laure: Modular state space analysis of coloured Petri nets. In: MICHELIS, Giorgio D. (Hrsg.) ; DIAZ, Michel (Hrsg.): *Application and Theory of Petri Nets 1995, 16th International Conference* Bd. 935. Turin, Italy, : Springer-Verlag, jun 1995, S. 201–217. – URL http://www.daimi.au.dk/CPnets/publ/full-papers/ChrPet1995.pdf. – URL besucht am 05. August 2006 19, 21

[Clarke und Schlingloff 2001] CLARKE, E.M. ; SCHLINGLOFF, H.: Model Checking. In: ROBINSON, A. (Hrsg.) ; VORONKOV, A. (Hrsg.): *Handbook of Automated Reasoning* Bd. II. Elsevier Science, 2001, Kap. 24, S. 1635–1790. – URL http://www.cs.cmu.edu/~emc/papers/Booksandbook-of-Automated-Reasoning_Clarke-Schlingloff_Model-Checking.ps. – URL besucht am 10. August 2006 10

[CollabNet 2006a] COLLABNET: *JavaCC [tm]: Grammar Files*. Online Dokumentation. 2006. – URL https://javacc.dev.java.net/doc/javaccgrm.html. – URL besucht am 12. September 2006 28, 40

[CollabNet 2006b] COLLABNET: *JavaCC [tm]: Lookahead Mini Tutorial*. Online Dokumentation. 2006. – URL https://javacc.dev.java.net/doc/lookahead.html. – URL besucht am 13. September 2006 40

[Dijkstra 1972] DIJKSTRA, Edsger W.: The Humble Programmer. In: *Communications of the ACM* 15 (1972), October, Nr. 10, S. 859–866. – URL http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html. – Turing Award Lecture 1

[Duvigneau 2002] DUVIGNEAU, Michael: *Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten*, Universität Hamburg, Department Informatik, Diplomarbeit, Dezember 2002 18

[Girault und Valk 2003] GIRAULT, C. ; VALK, Rüdiger: *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. 2003 18

[JavaPathFinder 2006] JAVAPATHFINDER: *JavaPathFinder*. 2006. – URL http://javapathfinder.sourceforge.net/. – Software 21

[Kindler und Völzer 1997] KINDLER, Ekkard ; VÖLZER, Hagen: Flexibility in algebraic nets / Humboldt-Universität zu Berlin. URL http://www.tcs.uni-luebeck.de/pages/voelzer/Publications/KiVo98.ps, November 1997 (89). – Informatik-Berichte. URL besucht am 04. August 2006 3, 7, 8, 9

[Kummer 2002] KUMMER, Olaf: *Referenznetze*. Berlin : Logos Verlag, 2002. – URL http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=deu&id=. – URL besucht am 04. August 2006. – ISBN 3-8325-0035-9 4, 5, 17, 18, 19

[Kummer u. a. 2006a] KUMMER, Olaf ; WIENBERG, Frank ; DUVIGNEAU, Michael: *Renew – The Reference Net Workshop*. Available at: http://www.renew.de/. Mai 2006. – URL http://www.renew.de/. – Software – Entwicklungsversion 2.2 3

[Kummer u. a. 2006b] KUMMER, Olaf ; WIENBERG, Frank ; DUVIGNEAU, Michael: *Renew – User Guide*. Release 2.1. Hamburg: University of Hamburg, Faculty of Informatics, Theoretical Foundations Group (Veranst.), Mai 2006. – URL http://www.renew.de/. – Available at: http://www.renew.de/ 5, 6, 17, 38

[Kummer u. a. 2004] KUMMER, Olaf ; WIENBERG, Frank ; DUVIGNEAU, Michael ; SCHUMACHER, Jörn ; KÖHLER, Michael ; MOLDT, Daniel ; RÖLKE, Heiko ; VALK, Rüdiger: An Extensible Editor and Simulation Engine for Petri Nets: Renew. In: CORTADELLA, Jordi (Hrsg.) ; REISIG, Wolfgang (Hrsg.): *Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings* Bd. 3099. Heidelberg : Springer, Juni 2004, S. 484–493. – URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3099&spage=484. – URL besucht am 04. August 2006 3

[Latvala 2001] LATVALA, Timo: Model Checking LTL Properties of High-Level Petri Nets with Fairness Constraints. In: COLOM, J-M. (Hrsg.) ; KOUTNY, M. (Hrsg.): *Application and Theory of Petri Nets*. Berlin : Springer, 2001, S. 242–262 12, 21

[Mäkelä 2001] MÄKELÄ, Marko: A Reachability Analyser for Algebraic System Nets / Helsinki University of Technology, Laboratory for Theoretical Computer Science. Espoo, Finland, June 2001 (A69). – Research Report. – 93 S. – URL http://www.tcs.hut.fi/Publications/info/bibdb.HUT-TCS-A69.shtml. URL besucht am 04. August 2006 8, 9, 11

[Mäkelä 2002] MÄKELÄ, Marko: Maria: Modular Reachability Analyser for Algebraic System Nets. In: ESPARZA, Javier (Hrsg.) ; LAKOS, Charles (Hrsg.): *Application and Theory of Petri Nets 2002: 23rd International Conference, ICATPN 2002*. Adelaide, Australia : Springer-Verlag, Berlin, Germany, June 2002 (Lecture Notes in Computer Science 2360), S. 434–444. – URL http://www.tcs.hut.fi/Publications/msmakela/maria.pdf. – URL besucht am 04. August 2006 3

[Mäkelä 2003] MÄKELÄ, Marko: *Efficient Computer-Aided Verification of Parallel and Distributed Software Systems*. Espoo, Finland, Helsinki University of Technology, Dissertation, nov 2003. – URL http://lib.tkk.fi/Diss/2003/isbn9512267926/. – URL besucht am 05. August 2006 11, 19

[Mäkelä u. a. 2005] MÄKELÄ, Marko ; LATVALA, Timo ; VARPAANIEMI, Kimmo: *Maria 1.3.5—Modular Reachability Analyser*. Teknillinen korkeakoulu, Tietojenkäsittelyteorian laboratorio (Helsinki University of Technology, Laboratory for Theoretical Computer Science), Espoo, Finland. July 2005. – URL http://www.tcs.hut.fi/Software/maria/. – Software 7, 8, 11, 17, 19, 40

[Schumacher 2003] SCHUMACHER, Jörn: *Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung*, Universität Hamburg, Department Informatik, Diplomarbeit, 2003 6

[Sun Microsystems 2006] SUN MICROSYSTEMS: *The Swing Tutorial*. Online Dokumentation. 2006. – URL http://java.sun.com/docs/books/tutorial/uiswing/index.html. – URL besucht am 13. September 2006 37

[Valk 2000] VALK, Rüdiger: Relating Different Semantics for Object Petri Nets, Formal Proofs and Examples / University of Hamburg, Department for Computer Science Report/00. URL http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/valk/Bericht226valk.pdf, 2000 (FBI-HH-B-226). – Technical Report. URL besucht am 11. August 2006 17

[Valk 2004] VALK, Rüdiger: Vorlesungsunterlagen zur Vorlesung „Prozesse und Nebenläufigkeit“ Wintersemester 2004/2005 / Universität Hamburg, Department Informatik. 2004. – Skript. Nicht frei verfügbar, die zitierten Sachverhalte sollten jedoch auch in anderen Grundlagenwerken zu finden sein. 10

[Valk und Moldt 2006] VALK, Rüdiger ; MOLDT, Daniel: Vorlesungsunterlagen zu F4, Sommersemester 2006 / Universität Hamburg, Department Informatik. 2006. – Skript. Nicht frei verfügbar, die zitierten Sachverhalte sollten jedoch auch in anderen Grundlagenwerken zu finden sein. 14, 16, 18