Abschlussbericht Projektseminar Modellbasierte Softwareentwicklung

Projektseminar eingereicht von
Nicolas Acero, Sebastian Ehmes, Huynh-Tan Truong, Paul Wiedenbeck
am 10. September 2017

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Echtzeitsysteme

Elektrotechnik und Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Prof. Dr. rer. nat. A. Schürr
Betreuer: Lars Fritsche, Erhan Leblebici

ES-B-0060

Erklärung zum Projektseminar

Hiermit versicheren wir, das vorliegende Projektseminar selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die wir aus fremden Quellen direkt oder indirekt übernommen haben, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Wir erklären uns damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 10. September 2017

(Nicolas Acero, Sebastian Ehmes, Huynh-Tan Truong, Paul Wiedenbeck)

Inhaltsverzeichnis

1 Aufgabenstellung

Modellbasierte Softwareentwicklung ist ein Konzept, bei dem Techniken angewendet werden, die aus formalen Modellen automatisiert Programmcode erzeugen. Innerhalb dieses Projektseminares werden Tripe Graph Grammatiken (TGGs) verwendet, welche Regeln spezifizieren, anhand derer Transformationen zwischen verschiedenen Metamodellen festgelegt werden.

Gegeben ist ein Netwerk, bestehend aus Servern, Computern und Routern, die über Kabel verbunden sind, wobei Server und Computer immer mit einem Router und nicht direkt miteinander verbunden sind. Es gibt zwei Klassen von Kabeln, Kupfer und Glasfaserkabel, die verschiedene Übertragungsraten ermöglichen. Es soll durch Angabe einer Spezifikation von Anforderungen (im Folgenden Requirements genannt) automatisiert überprüft werden, ob die Anforderungen im gegebenen Netzwerk erfüllbar sind.

Dieses für dieses Projektseminar vereinfachte Szenario simuliert eine Cloud Infrastruktur, bei der Kunden ein (Teil-) Netzwerk mieten können. Dabei wird aus Sicht des Kunden vom realen Netwerk abstrahiert, da für ihn nicht wichtig ist, auf welchen Serven genau seine Dienste ausgeführt werden bzw. über welche exakte Topologie diese verbunden sind. Stattdessen spezifiziert der Kunde eine Liste von Diensten (Provider) sowie eine Liste von sog. Consumern, also Prozesse, die auf die Dienste zugreifen.

Zunächst ist von uns eine Sprache zu entwickeln, die mächtig genug ist, diese Spezifikation anzugeben sowie, analog dazu, eine weitere Sprache, die zur Modellierung einer gegebenen Netzwerktopologie geeignet ist. Mittels dieser Sprachen lassen sich vom Anwender Anforderungen sowie vom Anbieter das existierende Netzwerk (im Kontext dieses Seminares als *Implementation* bezeichnet) in textueller Form beschreiben.

Aus diesen textuellen Beschreibungen sollen sich Ecore Modelle generieren lassen. Zwischen zwei Ecore Modellen (Requirements und Implementation) soll mittels Triple Graph Grammatiken eine Zuordnung gesucht werden. Dabei sollen auch eine Menge an Contraints berücksichtigt werden, die unter Anderem die maximale Kapazität eines Servers und die Länge des Pfads einer Verbindung berücksichtigen. Dabei gibt es funktionale Contraints sowie nichtfunktionale Contraints.

In der ersten Iteration werden 1-Hop Netzwerktopologien betrachtet, was hinsichtlich vereinfachter Testfälle Vorteile in der Implementierung der TGGs mit sich bringt. Um auch einen kürzesten Pfad als Optimierungsproblem betrachten zu können, werden in der zweiten Iteration auch Multihop Topologien verarbeitet.

2 Domain Specific Languages

Zur Spezifizierung von Requirements und Implementation (also gegebenem physikalischem Netzwerk) haben wir zwei Sprachen definiert. Dabei wurde das Framework *Xtext* [1] verwendet, was wiederum auf das *Eclipse Modelling Framework* [2] zurückgreift. Xtext generiert aus der Sprachdefinition einen Parser, ein Metamodell sowie ein Eclipse Plugin, mit dessen Hilfe man Dateien in der definierten Sprache erstellen und bearbeiten kann.

Das komfortable an Xtext ist, dass man innerhalb des generierten Eclipse Plugin, ein Syntax-Highlighting und eine Auto-Completion für seine selbst entworfene Sprache geliefert bekommt.

2.1 Implementation

Im Folgenden ist die Sprachendefinition auszugsweise beschrieben:

```
XContainer :
        elements+=NetworkElement*;
```

Jedes Gerät erbt von einer gemeinsamen Klasse, *XContainer*. Diese wird als initialer Ankerpunkt verwendet.

```
NetworkElement returns NetworkElements:
        XDevice | XCable;

XDevice returns XDevice:
        XServer | XComputer | XRouter;

XCable returns XCable:
        XCopperCable | XGlassFiberCable;
```

Es lassen sich Geräte, also *Server*, *Computer* und *Router* sowie Verbindungen mittels Kupfer und Glasfaserkabel definieren.

```
Connections:
        {Connections}
                ('incoming' '(' incoming+=[XCable]
                 (',' incoming+=[XCable])* ')' )?
                ('outgoing' '(' outgoing+=[XCable]
                 (',' outgoing+=[XCable])* ')' )?;
```

Ein Gerät kann eingehende und ausgehende Verbindungen besitzen. Eine Verbindung hat immer zwei *Endpunkte*, wobei die Endpunkte über ein Kabel festgelegt werden. Zwei Geräte, die verbunden sind, referenzieren das gleiche Kabel, wobei ein Gerät das Kabel als *incoming* und das andere Gerät als *outgoing* angibt.

```
CableAttributes:
        {CableAttributes}
                'speed' speed = EBigInteger
                'endPoint1' endPoint1= [XDevice]
```

```
        'endPoint2' endPoint2= [XDevice]
        ('isDuplex' isDuplex = bool)?;
```

Ein Kabel hat neben den Endpunkten, zwischen denen es verlegt ist, eine maximale *Datenrate* sowie das Attribut *isDuplex*, das angibt ob nur Daten von *endPoint1* zu *endPoint2* übertragen werden können oder in beide Richtungen.

```
XServer returns XServer:
        'Server'
        name=ID
        '{'
        'maxSlots' maxSlots=EBigInteger
        'MTBF' MTBF=EBigInteger
        (connections=Connections)
        '}';
```

Ein Server kann eine maximale Anzahl an Diensten (*Services*) betreiben (*maxSlots*). Die Zuverlässigkeit ist über das Attribut *MTBF* (Mean Time between Failure [3]) angegeben. Ein höherer Wert bedeutet eine höhere Zuverlässigkeit. Während *maxSlots* eine funktionale Anforderung darstellt, ist die *MTBF* ein weiches Constraint. Im späteren Verlauf des Projektseminars wird ein Optimierungsverfahren eingesetzt, das versucht, anhand dieser Attribute eine möglichst gute Zuordnung zwischen Requirements und Implementation zu finden. Was genau *möglichst gut* in diesem Kontext bedeutet, wird in Kapitel 4.2 beschrieben.

```
XComputer returns XComputer:
        'Computer'
        name=ID
        '{'
        (connections=Connections)
        '}';

XRouter returns XRouter:
        'Router'
        name=ID
        '{'
        'maxSpeed' maxSpeed=EBigInteger
        (connections=Connections)
        '}';
```

Computer und Router sind ähnlich aufgebaut wie Server, beide verwenden den gleichen Syntax um Verbindungen anzugeben. Router enthalten noch ein Constraint *maxSpeed*, das die maximal mögliche Summe an Datenraten aller aktiven Verbindungen ausdrückt.

2.2 Requirements

Die Requirements lassen sich in einer einfacheren Sprache ausdrücken, da diese keine Informationen über die Topologie enthalten.

```
XContainer :
    (agents+=XAgent)*;

XAgent returns XAgent:
        XProvider | XConsumer;
```

Es existieren *Provider* und *Consumer*, die beide Spezialisierungen von *XContainer* sind, ähnlich wie im Falle der Implementation DSL.

```
XProvider returns XProvider:
        'Provider'
        name=ID
        '{'
        'speed' speed=EBigInteger
        '}';

XConsumer returns XConsumer:
        'Consumer'
        name=ID
        '{'
        'speed' speed=EBigInteger
        '}';
```

Sowohl *Provider* als auch *Consumer* besitzen neben einem Namen eine Angabe über die benötigte Bandbreite. Eine beispielhafte Definition der Requirements mit 2 Providern und 3 Consumern könnte so aussehen:

```
Provider p1 {
        speed 4
}

Provider p2 {
        speed 5
}

Consumer c1 {
        speed 2
}

Consumer c2 {
        speed 2
}

Provider c3 {
        speed 2
```

```
}
```

3 Triple Graph Grammatiken

Triple Graph Grammatiken werden zur bidirektionalen (Meta-) Modelltransformation verwendet. Für dieses Seminar sind zwei Metamodelle (*Requirements* und *Implementation*) vorgegeben. Über die in Kapitel 2 vorgestellten Sprachen lassen sich für jede der zwei Seiten textuelle Beschreibungen verwenden. Aus einer solchen Beschreibung kann Xtext automatisch ein Modell erzeugen.

Obwohl beide Modelle ähnlich aufgebaut sind, ist das von Xtext erzeugte Metamodell nicht kompatibel zu dem vorgegebenen Metamodell. Um aus einer textuellen Beschreibung ein Modell, das dem gegebenen Format entspricht, erzeugen zu können, werden *TGGs* verwendet. Dabei wird eine Menge an Regeln spezifiziert, die die Transformation von Xtext Modell zu vorgegebenen Modell beschreibt.

3.1 Implementation

Der Hauptunterschied zwischen dem vorgegebenen und dem Xtext Metamodell *Implementation* besteht in der Semantik der Verbindungen bzw. Kabeln. Das vorgegebene Metamodell weist einem Kabel grundsätzlich eine Richtung zu, Verbindungen sind also immer unidirektional. Um Duplexverbindungen zu modellieren, ist es notwendig, zwei Kabel zu erstellen. Das Xtext Metamodell hingegen unterstützt nativ Duplexverbindungen, was den Modellierungsaufwand verringert. Abbildung 3.1 vergleicht beide Modelle hinsichtlich der Modellierung eines Kabels.

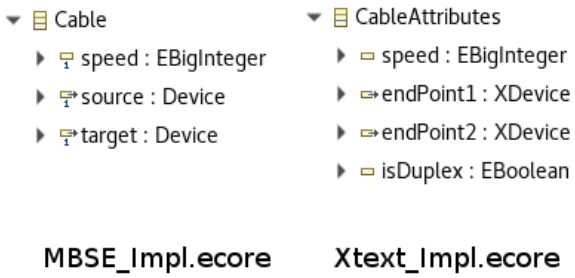

Abbildung 3.1: Vergleich der Kabelmodellierung zwischen Xtext Metamodell und gegebenem Metamodell

Beim Transformieren von *Source* (Xtext Modell) zu *Target* (gegebenes Modell) wird daher für jedes Kabel geprüft, ob es sich um eine Duplexverbindung handelt.

```
#source {
...
        ++ cableAttributes : CableAttributes {
                isDuplex := true
        }
}
```

Es existieren zwei kabelbezogene Regeln, je eine für Simplex und eine für Duplexverbindungen. Abhängig vom Attribut *isDuplex* wird entweder die Simplexregel oder die Duplexregel angewendet.

```
#target {

        container : Container {
                        -devices->device
                ++      -cables->cable1
                ++      -cables->cable2
        }

        device : Device

        ++ cable1 : Cable
        ++ cable2 : Cable

}
```

Für Duplexverbindungen werden zwei Kabel erzeugt, während die Simplexregel an dieser Stelle nur ein Kabel zuordnet bzw. anlegt. Bis auf diesen Unterschied sind beide Regeln gleich.

3.2 Requirements

Die Metamodelle auf der *Requirements*-Seite ähneln sich mehr als die auf *Implementation*-Seite. Die *TGGs* beschränken sich hier hauptsächlich auf Zuordnen der Elemente der verschiedenen Namensräume. Die Benennung innerhalb des Xtext Metamodells ist dabei so gewählt, dass sie der Benennung des gegebenen *Requirements*- Metamodells entspricht; jedoch ist jeder Bezeichnung ein X vorangestellt. Ein *xContainer* entspricht also einem *Container*, ein *XProvider* einem *Provider* usw.

4 Matching zwischen Requirements und Implementation

Aus zwei *xmi*-Dateien, *Requirements* und *Implementation*, welche Instanzen der Requirements bzw. Implementation Metamodelle beschreiben, soll nun eine geeignete Zuordnung gefunden werden, falls eine solche existiert. Berücksichtigt werden sollen funktionale Anforderungen, wie bspw. dass ein Provider immer einem Server und nicht einem Router zugeordnet wird oder dass die maximale Anzahl an verfügbaren Slots nicht überschritten wird. Zusätzlich sollen Kriterien wie die *MTBF* und auch die Pfladlänge innerhalb des Netzwerks in die Suche nach dem optimalen Matching mit einfließen.

Das Matching ist über ein mehrstufiges Verfahren implementiert:

1. Zuerst wird mittels *TGG* Regeln eine Menge an möglichen Matches gebildet, ohne Contraints zu berücksichtigen. Dabei werden, im Unterschied zu den TGGs, die zwischen DSL und gegebenem Metamodell übersetzen, keine neuen Elemente hinzugefügt, sondern lediglich nach Korrespondenzen gesucht.

2. Danach wird per *Integer Linear Programming* [4] eine Zielfunktion (*objective function*), welche die Constraints ausdrückt, maximiert. Dieser Schritt wird in das externe Tool *Gurobi* [5] ausgelagert, wodurch eine sehr hohe Geschwindigkeit erreicht wird. Während diesem Schritt werden alle Zuordnungen verworfen, die Constraints verletzen, beispielsweise durch Überschreiten von *maxSlots*. Bis zu diesem Schritt wird von der Netzwerktopologie abstrahiert. Die Schritte eins und zwei stellen die Basisimplementierung dar (Umsetzung 1).

3. Schließlich wird hinsichtlich des Pfades innerhalb des Netzwerks optimiert. Dieser Schritt ist in Umsetzung 2 implementiert. Dabei wird die Anzahl der Router (*Hops*) berücksichtigt, die zwischen einem Server und dem zugehörigem Computer liegen. Eine geringe Anzahl an Hops wird bevorzugt, da sich dadurch die Latenz verringert und mit höherer Wahrscheinlichkeit eine effizientere Nutzung der Netzwerkinfrastruktur (Router- und Kabelauslastung) erzielt wird als bei langen Pfaden. Die Suche nach dem kürzesten Pfad bedient sich einem Hilfsmittel. Unter Verwendung eines zusätzlichen Knotentyps (*VirtualNode*) wird die Netzwerktopologie berücksichtigt (Siehe Kapitel 4.3).

4.1 Umsetzung 1 - Korrespondenzsuche

Mittels TGG basierter Regeln wird eine allgemeine Menge an erlaubten Zuordnungen erstellt. An diesem Punkt werden die Constraints noch nicht berücksichtigt, die entstehende Menge ist also relativ groß. Mittels sogenannter *Ignore-Rules* werden alle Elemente auf Implementation Seite registriert, aber keine Korrenspondenz gesucht. Als Beispiel ist hier die Regel abgebildet, die Kabel auf Implementation Seite behandelt:

```
#source {
  reqContainer : requirements.Container
}
```

```
#target {
  implContainer : implementation.Container {
    ++ -cables->cable
  }
  ++ cable : implementation.Cable
}
```

Schließlich wird noch Kontext angegeben:

```
#correspondence {
  reqContainerToImplContainer : ReqContainerToImplContainer {
     #src->reqContainer
     #trg->implContainer
  }
}
```

Durch Verwendung der *Ignore-Rules* sind die Elemente auf Implementation Seite nun als Kontext bekannt. Jetzt werden *Provider* auf *Server* und *Consumer* auf *Computer* gemapt. Nachfolgend ist als Beispiel die Regel angegeben, die *Provider* zu *Server* zuordnet:

```
#source {
  ++ reqAgent : requirements.Provider
  }

  #target {
    implDevice : implementation.Server {
      -outgoing->cable
    }

    cable : implementation.Cable
  }

  #correspondence {
    ++ reqAgentToImplDevice : ReqAgentToImplDevice {
      #src->reqAgent
      #trg->implDevice
    }
  }

  #attributeConditions {

  }
```

Da zu diesem Zeitpunkt alle *Cable* und *Devices* durch die *Ignore-Rules* behandelt wurden, können in dieser Regel **implDevice** und **cable** als bereits vorhandener Kontext

vorausgesetzt werden. Die richtige Reihenfolge der Ausführung der Regeln ergibt sich automatisch, da die Regeln, die Kontext voraussetzen, von eMoflon nicht angewendet werden können, wenn dieser Kontext noch fehlt. Die *Ignore-Rules* können jedoch immer angewendet werden.

4.2 Umsetzung 1 - Integer Linear Programming

Die Constraints werden durch mehrere *Objective Functions* ausgedrückt. Sie haben die Form:

$$f(v_i) = \alpha * v_1 + \beta * v_2 + \gamma * v_3 + ... \tag{1}$$

wobei v_i jeweils 1 oder 0 ist. α, β, γ bezeichnen Gewichte. Zum Lösen der Gleichungen verwenden wir *Gurobi* [5]. Gurobi bietet den Vorteil, dass es eine Javaschnittstelle anbietet und sich direkt aus der von eMoflon generierten Codebasis ansprechen lässt. Hier wird für jeden *Server* ein Constraint angelegt:

```
for(Server s : idToCoefficientMap.keySet()) {
  results.add(new UserDefinedILPConstraint(
    idToCoefficientMap.get(s),
    "<=",
    s.getMaxSlots().doubleValue()));
}
```

MaxSlots() darf hier nicht überschritten werden. Jedes Mapping auf einem Server hat zuvor das Gewicht 1 erhalten, sodass ein Provider jeweils einen Slot verbraucht.

4.3 Umsetzung 2 - Berücksichtigung der Netzwerktopologie

In der ersten Umsetzung ist es möglich unter Einhaltung von Nebenbedingungen wie zum Beispiel, dass ein Server nur begrenzt Provider aufnehmen kann oder ein Server mit hoher mittleren Betriebsdauer zwischen Ausfällen (MTBF) bevorzugt wird, verschiedene Provider und Consumer auf die passenden Netzwerkressourcen zu mappen. Die Umsetzung 2 erweitert das Mapping durch Berücksichtigung der Netzwerktopologie, das heißt ein Server wird bevorzugt gewählt, wenn er wenig *Hops* bis zu einem Computer aufweist, das entspricht dem kürzeseten Pfad im Netzwerk.

4.3.1 Idee

Das implementierte Netzwerk wird auf einen *Dummy Knoten* (der VirtualNode) gemappt. Beim Mappen wird ein *Hop* (der Pfad von einem Router zu einem anderem Router) negativ gewichtet. Daraus folgt, dass die Parameter eines gesamten Pfades in der Objectivefunction negative Werte animmt. Da die Objectivefunction beim ILP maximiert wird, folgt daraus dass der kürzeste Pfad von einem Server zu Computer als Ergebnis zurückgeliefert wird. Abzulesen ist dieser dann vom VirtualNode aus, da nach der Optimierung nur noch Korrespondenzen zwischen VirtualNode und einem Netzwerkelement vorhanden sind, die zum kürzesten Pfad gehören.

4.3.2 Regelmenge (TGGs)

Zur Umsetzung werden im Prinzip nur 4 Regeln benötigt. Die erste Regel (Abbildung 4.1) registriert den VirtualNode:

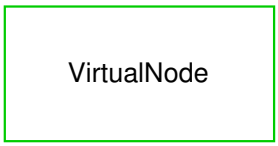

Abbildung 4.1: Regel eins: Erzeugung des VirtualNodes

Die zweite Regel (Abbildung 4.2) erzeugt eine Korrespondenz zwischen dem VirtualNode und einem Router:
Diese Regel wird genau ein einziges Mal ausgeführt, da wir einen Startknoten benötigen

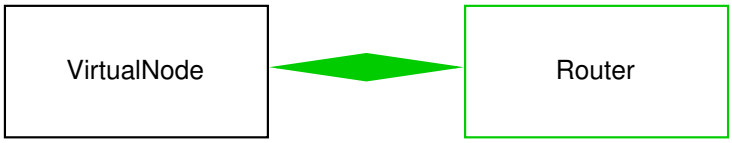

Abbildung 4.2: Regel zwei: Korrespondenz VirtualNode und Router

um den kürzesten Pfad zu finden. Die dritte Regel (Abbildung 4.3) realisiert das Mapping der *Hops* durch erzeugen von Korrespondenz zwischen dem VirtualNode und einem Router, der mit einem registrierten Router verbunden ist. Diese Korrespondenz erhält ein

negatives Gewicht um bei der Optimierung den kürzesten Pfad zu erhalten.
Die vierte Regel (Abbildung 4.4) erzeugt eine Korrespondenz zwischen VirtualNode und

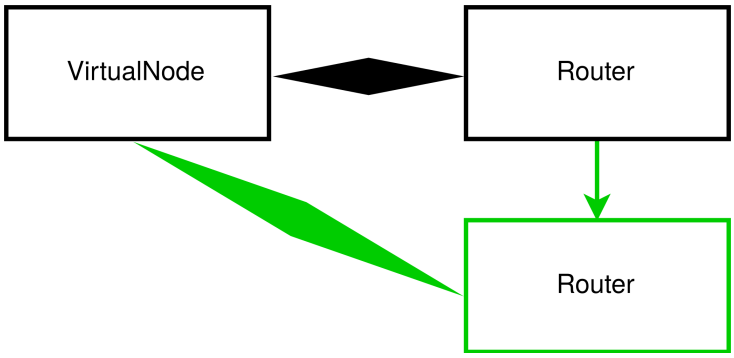

Abbildung 4.3: Regel drei: Korrespondenz VirtualNode, bereits registriertem Router und Router

einem Gerät (Server/Computer), der mit einem registrierten Router verbunden ist.

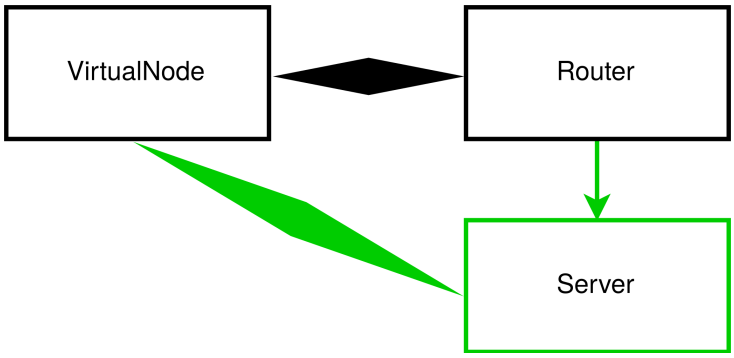

Abbildung 4.4: Regel vier: Korrespondenz VirtualNode, bereits registriertem Router und Gerät, gezeigt am Beispiel der Regel für das Hinzufügen einer Server-Korrespondenz.

5 User Interface

Erweitert wurde die Aufgabe durch Entwicklung einer graphischen Oberfläche zur Ausführung der entwickelten Software. Die Idee zur Entwicklung einer graphischen Oberfläche ist, dass das Laden der Metamodelle und Ausführen der Transformation einige Schritte, wie z.B. das Verschieben und/oder Umbenennen von generierten Dateien, in vielen verschiedenen Pakete benötigt hat. Für eine Person, die sich nicht in der entwickelten Software auskennt, ist die Ausführung der Transformation ein schwerer Prozess. Zusätzlich ist das Ergebnis der Transformation in Textform sehr abstrakt, daher ist die Visualisierung vom Ergebnis sehr nützlich. Die graphische Oberfläche erlaubt es die Metamodelle für Requirement und Implementation zu Laden um anschließend das Mapping durchzuführen. Dabei wird die Netzwerktopologie auf Implementations-Seite und das gefundene Mapping visualisiert. Nachfolgend ist ein Beispiel Workflow beschrieben. Wie man in Abb. 5.1 erkennen kann, beginnt der Workflow mit dem Erstellen einer Instanz des Requirements Metamodells, mit Hilfe unserer neu Entwickelten DSL. Durch Abspeichern wird automatisch eine Instanz des DSL-Metamodells erzeugt, die unseren Entwurf verkörpert. Danach wird, ebenfalls automatisch, die TGG ausgelöst, die das erzeugte Modell in eine Instanz des für das Mapping benötigten Requirements Metamodells transformiert. Abbildung 5.2 zeigt analogen den Prozess der Implementation-Seite und damit den nächsten Schritt des Workflows. In beiden Fällen kann man die erzeugten Modelle in unserer bereitgestellten GUI anzeigen lassen und so beispielsweise auf Fehler überprüfen. Um diesen Prozess weiter zu vereinfachen, werden die Resultate der TGG automatisch in einem Ordner des Projektes der Benutzeroberfläche gespeichert. Der letzte Schritt im Workflow kann vollständig in unserer Benutzeroberfläche durchgeführt werden. Hierfür werden, wie in Abb. 5.3 gezeigt, zunächst ein Requirements-Models als *Source* und ein Implementation-Model als *Target* der gesuchten Korrespondenz ausgewählt. Durch Klicken auf *find Mapping* wird die Korrespondenz zwischen beiden Modellen gesucht und anschließend gespeichtert. Durch *Load Models* wird die Liste der verfügbaren Modelle erneuert. Abb. 5.3 zeigt das gesuchte Modell der Korrespondenz (zu erkennen an dem Suffix *-corr*).

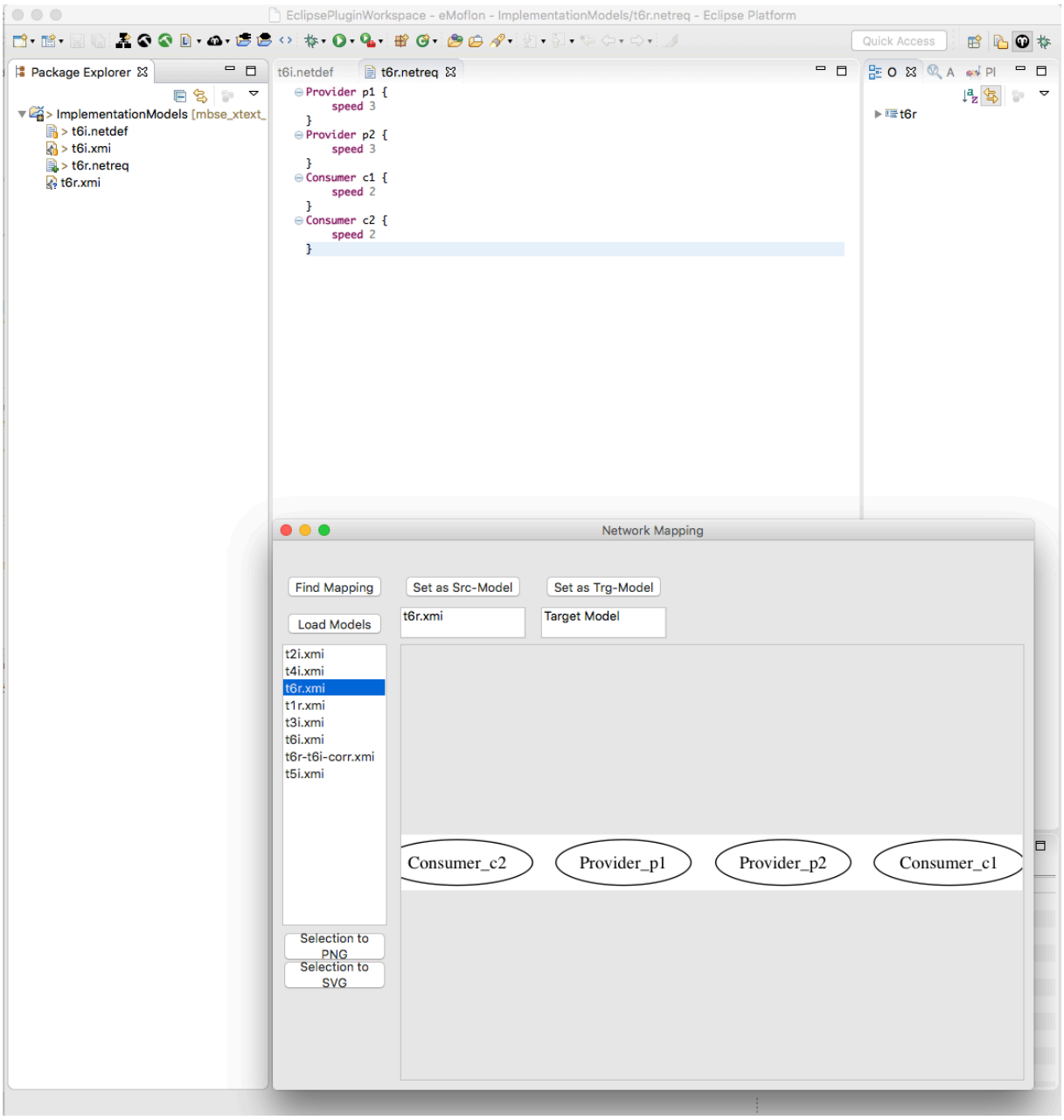

Abbildung 5.1: Erstellen der Requirements

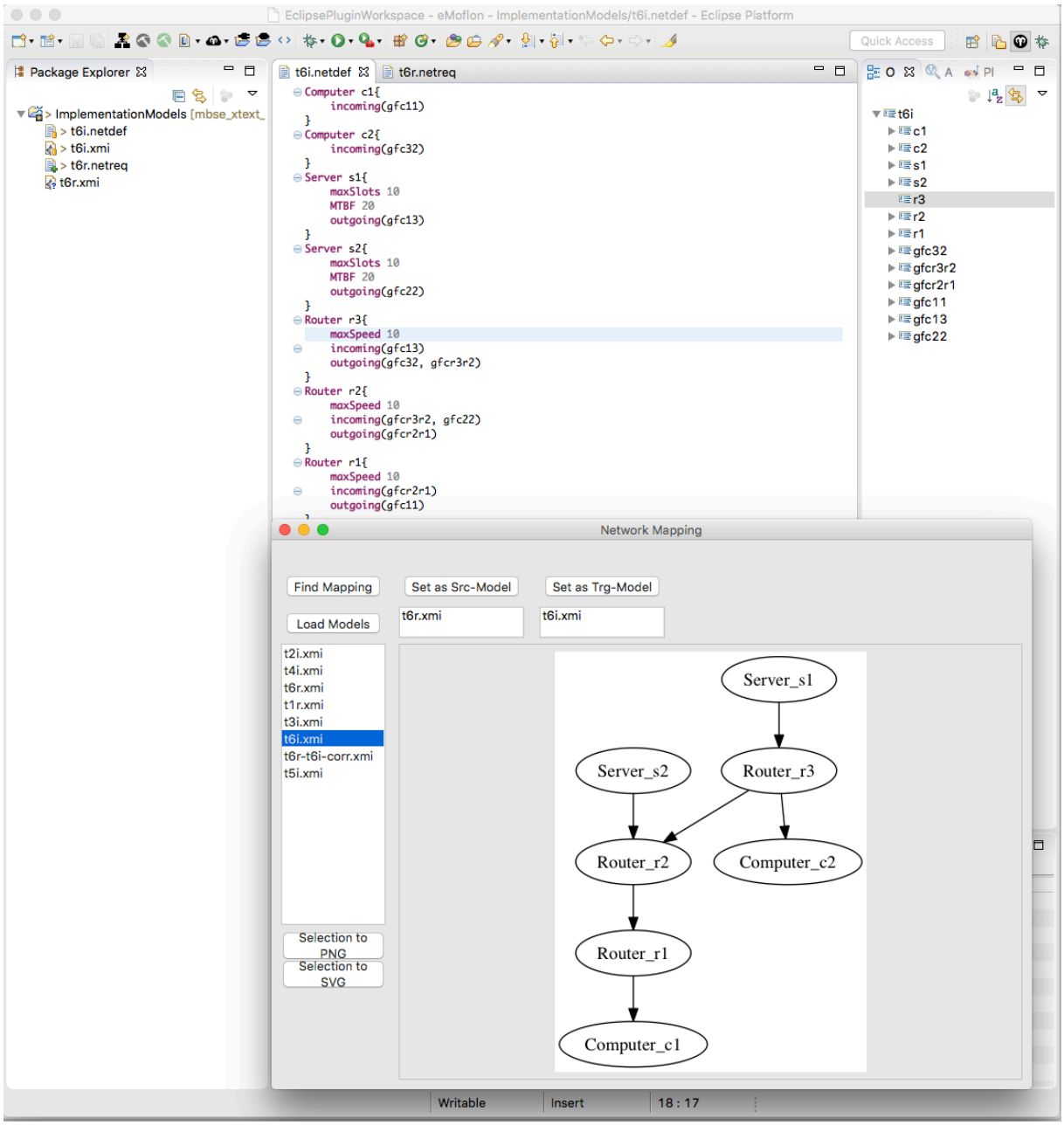

Abbildung 5.2: Erstellen der Spezifikation der Implementation

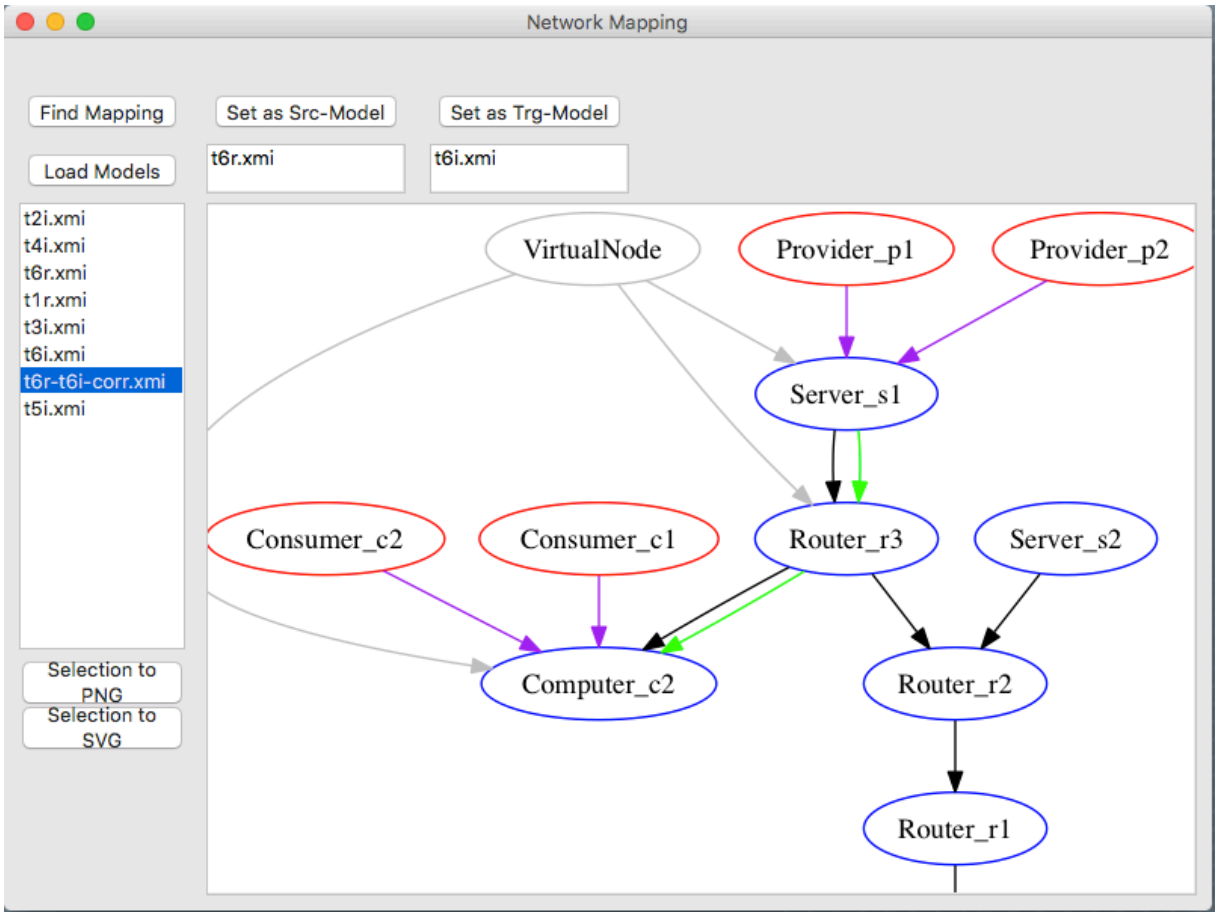

Abbildung 5.3: Ausgabe der Korrelation

6 Fazit

Ziel des Projektseminars *Modellbasierte Softwareentwicklung* ist es eine Abbildung von einer Requirement Spezifikation (Requirement) auf reale Ressourcen eines Netzwerks (Implementation) zu finden unter Anwendung von TGGs.

Im Rahmen dieses Projektseminars wurde eine DSL mit Xtext entwickelt um Requirement- und Implementation-Modelle zu beschreiben. Zusätzlich ist es notwendig gewesen TGGs zu definieren, die die DSL-Modelle in passende Metamodelle für eMoflon transformiert. Es wurden weitere TGGs geschrieben zum Mapping von Requirement-Spezifikationen auf Implementation-Ressourcen. Hierbei gibt es zwei Ansätze. Der erste Ansatz findet ein Mapping von der Requirement-Spezifikation, ungeachtet der Netzwerktopologie, zu den Implementation-Ressourcen unter Berücksichtigung von gegebenen Nebenbedingungen. Der zweite Ansatz erweitert den ersten Ansatz um die Berücksichtigung der Netzwerktopologie. Das Mapping von einem Consumer auf einen Server mit kürzesten Pfad zu einem Computer wird bevorzugt. Allerdings geht dabei die Fähigkeit zur Aufteilung, von Providern auf mehrere Server bzw. von Consumern auf mehrer Computer, verloren. Es wird in Folge dessen der Server, der die meisten Provider beherbergen kann mit dem Computer, der die meisten Consumer versorgen kann, auf dem kürzesten Weg verbunden. Alle Agenten, die nicht auf einer der beiden Netzwerkressourcen unterzubringen sind, werden verworfen. Wir vermuten, dass die Ursache hierfür in dem Constraint der ersten Regel (Korrespondenz zwischen Router und VirtualNode) und der Tatsache, dass die Geräte in den getesteten Topologieen alle durch undirektionale Kanten verbunden waren, liegt. Dadurch, dass diese Regel nur einen einzigen Router als Startpunkt für einen Pfad zulässt, kann in einer Topologie, in der alle Verbindungen unidirektional sind, nur genau eine Verbindung zwischen Server und Computer gefunden werden. Zur einfachen Nutzung der entwickelten Software wurde zusätzlich eine graphische Oberfläche implementiert. Die graphische Oberfläche ermöglicht es über einfaches Klicken Modelle zu laden um anschließend die Transformation durchzuführen. Zusammengefasst ist die Aufgabenstellung erfüllt mit der Erweiterung einer graphischen Oberfläche für die Benutzung der Software.

Literatur

[1] XText, a framework for developing domain specific languages, Homepage: https://eclipse.org/Xtext/

[2] Eclipse Modelling Framework, a modeling framework and code generation facility for building tools and other applications based on a structured data model, Homepage http://www.eclipse.org/modeling/emf/

[3] Mean Time Between Failure bezeichnet die mittlere Betriebsdauer zwischen Ausfällen, definiert in DIN EN/IEC61709

[4] Laurence A. Wolsey (1998). Integer programming. Wiley. ISBN 978-0-471-28366-9.

[5] Gurobi, ein Löser für mathematische Programmierung, Homepage: http://www.gurobi.com