Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Nebal El Bebbili

Ein graphischer Editor für Zustandsautomaten und dessen Integration in OMNeT++

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Nebal El Bebbili

Ein graphischer Editor für Zustandsautomaten und dessen Integration in OMNeT++

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Franz Korf
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 4. Mai 2016

Nebal El Bebbili

Thema der Arbeit
Ein graphischer Editor für Zustandsautomaten und dessen Integration in OMNeT++

Stichworte
OMNeT++, GMF, EMF, Eclipse Plugin, XSL Transformation, Zustandsautomat, Graphischer Editor, Codegenerierung

Kurzzusammenfassung
Die vorliegende Bachelorarbeit befasst sich mit der Visualisierung des Verhaltens der OMNeT++ Komponenten auf Basis von Finite State Machines. Für die Modellierung und die Darstellung der FSMs wird ein Eclipse Plugin entwickelt, welches sich in die OMNeT++ Simulationsumgebung integrieren lässt. Das Plugin stellt einen graphischen Editor zur Verfügung, der auf Basis von GMF entwickelt wurde. Mit diesem Editor lassen sich FSMs modellieren. Außerdem wird aus den modellierten FSMs auf Basis von XSLT ein Implementierungscode generiert. Anschließend werden die modellierten FSMs zur Laufzeit in der OMNeT++ Simulation dargestellt und aktualisiert.

Nebal El Bebbili

Title of the paper
A graphical editor for finite state machines and its integration into OMNeT ++

Keywords
OMNeT++, GMF, EMF, Eclipse Plugin, XSL Transformation, Finite State Machines, Graphical Editor, Code generation

Abstract
The bachelor thesis deals with the visualization of the behavior of the OMNeT++ components based on Finite State Machines. For modeling and displaying the FSMs an Eclipse plug-in is developed, which can be integrated into the simulation platform OMNeT++. The plugin provides a graphical editor that had been developed on the basis of GMF. With the help of the Editor FSMs can be modeled. From the modeled FSMs the implementation code will be generated with the help of XSLT. Finally the modeled FSMs will be displayed and updated during the runtime of the OMNeT++ simulation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Diese Arbeit ist in der Communication over Realtime Ethernet (CoRE) Arbeitsgruppe [CoRE Research Group] an der HAW Hamburg entstanden. Diese Arbeitsgruppe befasst sich mit der Realisierung von Echtzeit-Ethernet-Kommunikation im Auto und Flugzeug.

Der nahezu exponentiell wachsende Software- und Elektronik-Anteil der modernen Fahrzeuge wird durch die Funktionszunahme in Richtung Sicherheit und Zuverlässigkeit getrieben. Etwa 90% der Innovationen in Fahrzeugen betreffen den Elektronikbereich. Somit werden die Fahrzeuge mit ihrem hohen Anteil an elektronischen Komponenten zu sehr komplexen Systemen [vgl. Schäuffele und Zurawka, 2006]. Um das Verständnis der komplexen Systeme zu erleichtern, gibt es eine Menge von Ansätzen. Beispielsweise wurden Finite State Machines (FSMs) entwickelt, um komplexe Systeme leicht lesbar zu beschreiben. Daher ist der Einsatz von FSMs in solchen Systemen ein relevantes Hilfsmittel. Diese ermöglichen die Überprüfung des Verhaltens der einzelnen Systemkomponenten, um beispielsweise Systemfehler aufdecken zu können.

Ziel der Arbeit

Ziel dieser Arbeit ist es, Nutzern die Möglichkeit zu geben, das Verhalten von OMNeT++ Komponenten mit Hilfe von FSMs zu beschreiben. Dies setzt sich aus mehreren Teilen zusammen. Zunächst muss ein graphischer Editor zur Modellierung von FSMs entwickelt werden. Dann muss aus der modellierten FSM automatisch ein Implementierungscode generiert werden. Anschließend muss die modellierte FSM zur Laufzeit dargestellt und aktualisiert werden. Die graphische Darstellung der FSM zur Laufzeit ermöglicht es dem Nutzer, mit Hilfe der Aktualisierung das Verhalten einer OMNeT++ Komponente zu beobachten.

Aufbau der Arbeit

Kapitel 2 befasst sich mit den Grundlagen, die zum Verständnis der nachfolgenden Kapitel dienen. In diesem werden die Simulationsumgebung OMNeT++, Eclipse Plugins, Grundlagen zur Codegenerierung und zu den Finite State Machines einschließlich der OMNeT++ Finite State Machine näher erläutert. In Kapitel 3 werden die Anforderungen an die verschiedenen Ebenen der Arbeit definiert. Anschließend folgt in Kapitel 4 das Konzept der Arbeit. In Kapitel 5 wird die Implementierung des Konzepts beschrieben. Daraufhin wird in Kapitel 6 das entwickelte Plugin evaluiert und zum Schluss wird die gesamte Arbeit zusammengefasst.

2. Grundlagen

In diesem Kapitel werden für die vorliegende Arbeit relevanten Grundlagen vermittelt, die für das Verständnis der Architektur erforderlich sind. In Abschnitt 2.1 werden die Grundlagen der Simulationsumgebug OMNeT++ dargestellt. Danach werden in Abschnitt 2.2 die Entwicklungsumgebung Eclipse und deren Plugins erläutert. Im darauffolgenden Abschnitt 2.3 werden die grundlegenden Informationen zur Codegenerierung erklärt. Anschließend werden die allgemeine sowie die OMNeT++ spezifische Finite State Machine beschrieben (siehe Abschitt 2.4).

2.1. OMNeT++

Objective Modular Network Testbed in C++ (OMNeT++) [OMNeT++ Community, a] ist eine ereignisdiskrete Simulationsumgebung für die Modellierung von Kommunikationsnetzwerken, Multiprozessor- und verteilten Systemen. Diese basiert auf der C++ Programmiersprache und kann auf verschiedenen Betriebssystemen (Linux, Windows und Mac OS) genutzt werden [vgl. Varga und Hornig, 2008].

2.1.1. Modellstruktur

Die OMNeT++ Simulation besteht aus Modulen, die miteinander über Nachrichten kommunizieren. Es gibt zwei verschiedene Arten von Modulen: Simple und Compound Module. Ein Compound Modul enthält Untermodule. Aus diesem Grund ist die Anzahl der Hierarchieebenen unbegrenzt. Das gesamte Modell wird als Netzwerk bezeichnet, das wiederum auch ein Compound Modul ist. Die Topologie solcher Netzwerke wird in der Network Description (NED) Sprache festgelegt. Die zugehörige Logik der einzelnen Komponenten wird in C++ programmiert. In Abbildung 2.1 wird ein Netzwerk dargestellt, das aus einem Compound Modul und drei Simple Modulen besteht. Die kleinen Kästchen

stellen die Gates dar und verbinden die Module mit Hilfe von Connections (die Pfeile in Abbildung 2.1) miteinander. Nachrichten können entweder über Connections oder direkt zu anderen Modulen gesendet werden. Jedes Modul hat eine handleMessage() Methode zur Verarbeitung von ankommenden Nachrichten. [vgl. Manual OMNeT++ Community, b].

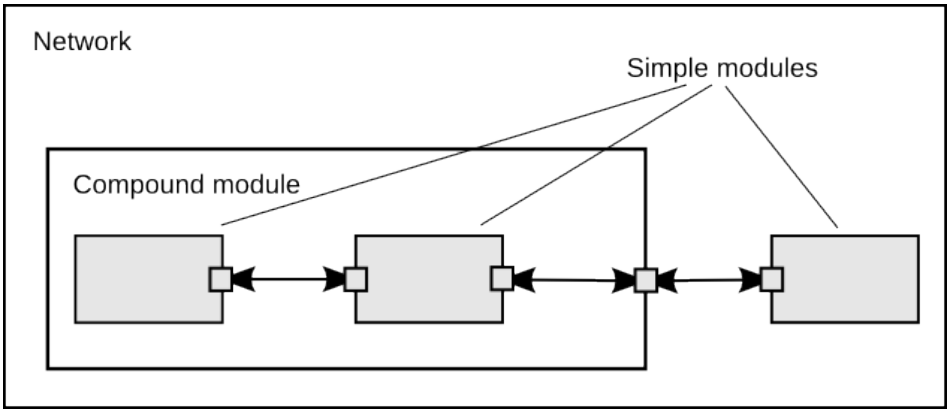

Abbildung 2.1.: Simple und Compound Module [Manual OMNeT++ Community, b, Abschnitt 2.1]

2.1.2. NED-Sprache

Die Network Description (NED) Sprache beschreibt die Struktur des zu simulierenden Netzwerks. Mit dieser kann der Benutzer beispielsweise festlegen, aus welchen Modulen das Netzwerk besteht und wie die einzelnen Komponenten miteinander verbunden sind. In der NED-Sprache werden auch Parameter unterstützt. Diese entsprechen Variablen, die zu einem Modul gehören, und können in der NED-Datei übergeben werden [vgl. Manual OMNeT++ Community, b].

Beispiel

Das Beispiel in Abbildung 2.2 zeigt ein Netzwerk *net*, das aus zwei Modulen *a* und *b* besteht. Diese sind mit Hilfe von Gates und Connections miteinander verbunden.

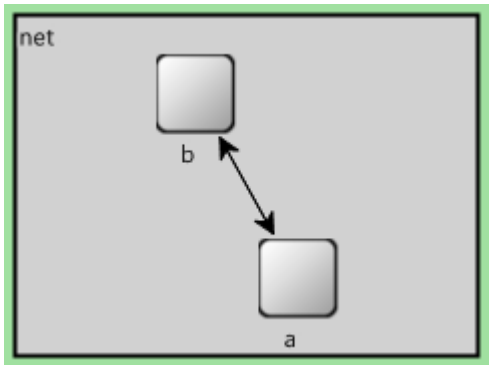

Abbildung 2.2.: Ein Simulationsnetzwerk *net*, bestehend aus zwei Modulen *a* und *b*

In Listing 2.1 wird der NED-Code für das Beispiel in Abbildung 2.2 erläutert. Als erstes wird das Simple Modul *mod* definiert, das die Inputs und Outputs der Gates enthält (Zeile 2-9). Danach folgt die Definition des Netzwerkes *net* (Zeile 12-21), das die zwei Untermodule *a* und *b* enthält, die vom Typ *mod* sind. Schließlich werden die zwei Untermodule mit Hilfe von deren Gates *in* und *out* über zwei Connections verbunden (Zeile 18 und 20).

```
1  //Definition des Simple Moduls mod
2  simple mod
3  {       //durch den Parameter @display wird das Aussehen des Moduls mod beschrieben
4          parameters:
5                  @display("i=block/routing");
6          gates:
7                  input in;
8                  output out;
9  }
10
11 //Definition des Netzwerkes net
12 network net
13 {       //Module zum Netzwerk hinzufuegen
14         submodules:
15                 a: mod;
16                 b: mod;
17         //Die Module a und b ueber Connections miteinander verbinden
18         connections:
19                 a.out --> b.in;
20                 a.in <-- b.out;
21 }
```

Listing 2.1: NED-Code für das Netzwerk in Abbildung 2.2

2.1.3. Definition eines Moduls in C++

Simple Module sind die aktiven Komponenten eines Simulationsmodells, in dem mit Hilfe der OMNeT++ Klassenbibliothek die Logik des Simulationsmodells festgelegt wird. Das Simulationsmodell wird aus Modulen und Verbindungen zusammengesetzt:

- **Module**: Das Verhalten eines Simple Moduls wird vom Benutzer in einer C++ Klasse, die von *cSimpleModule* vererbt werden muss, definiert.
- **Connections**: Diese können Channels sein und deren Verhalten kann ebenso in C++ festgelegt werden. Dafür muss der Benutzer von der *cChannel* Klasse oder einer der *cChannel* Unterklassen erben.

Folgende Abbildung zeigt das Verhältnis der oben genannten Klassen *cSimpleModule* und *cChannel*:

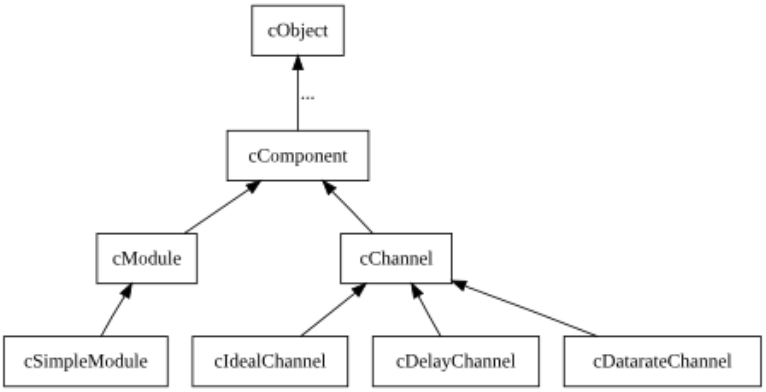

Abbildung 2.3.: Vererbung der Klassen *cComponent*, *cModule* und *cChannel* [Manual OMNeT++ Community, b, Abschnitt 4.2]

Simple Module und Channels werden durch die Redefinition einer bestimmten Menge von Funktionen programmiert. Einige dieser wichtigsten Funktionen befinden sich in der Basisklasse *cComponent*:

- ***initialize():*** Diese Funktion wird aufgerufen, nachdem die Konfiguration des Netzwerkes abgeschlossen ist.

- ***finish():*** Diese Funktion wird aufgerufen, wenn die Simulation erfolgreich beendet wird.
- ***handleMessage():*** Diese Funktion wird beim Eintreffen einer Nachricht aufgerufen, um diese zu verarbeiten.
- ***activity():*** Wird als kooperativ geschedulter Thread am Anfang der Simulation gestartet und läuft bis zum Ende der Simulation.

Beispiel

In Listing 2.2 ist ein einfaches Beispiel einer Definition eines Moduls in C++ zu sehen. Mit dem Aufruf *Define_Module(HelloModule)* wird die Klasse *HelloWorld* bei OMNeT++ als Modul registriert (Zeile 12). Das Beispiel redefiniert die Funktionen *initialize()* und *handleMessage()* der Basisklasse *cSimpleModule*. Dieses Modul hat zwei Aufgaben, nämlich am Anfang der Simulation „Hello World!“ auszugeben (Zeile 16) und die empfangenen Nachrichten zu löschen (Zeile 21).

```
1  // file: HelloModule.cc
2  #include <omnetpp.h>
3
4  class HelloModule : public cSimpleModule
5  {
6          protected:
7          virtual void initialize();
8          virtual void handleMessage(cMessage *msg);
9  };
10
11 //Register module class with OMNeT++
12 Define_Module(HelloModule);
13
14 void HelloModule::initialize()
15 {
16         ev << "Hello World!\n";
17 }
18
19 void HelloModule::handleMessage(cMessage *msg)
20 {
21         delete msg; //Just discard everything we receive
22 }
```

Listing 2.2: Beispiel einer Definition eines Simple Moduls in C++ [Manual OMNeT++ Community, b, Abschnitt 4.3]

2.2. Eclipse Plugin

Eclipse [Eclipse Foundation, a] ist eine Open-Source-Entwicklungsumgebung zur Entwicklung von Software nahezu aller Art. Grundsätzlich wurde diese für die Programmiersprache Java entworfen. Mittlerweile werden eine Menge von Sprachen (z.B. C++,

Scala, PHP usw.) mit Hilfe von Plugins unterstützt. Plugins sind Softwarepakete, die zur Erweiterung von Eclipse eingesetzt werden [vgl. Shavor, Sherry u. a., 2004, S. 200]. In dieser Arbeit wird grundsätzlich das Graphical Modeling Framework (GMF) [Eclipse Foundation, d] Plugin eingesetzt. Das GMF Plugin ist ein Open-Source-Framework, das einen modellgetriebenen Ansatz zur Entwicklung eines Modellierungwerkzeugs (grafischer Editor) in Eclipse bereitstellt. Im Rahmen dieser Arbeit wird ein graphischer Editor für die Modellierung von Finite State Machines (FSMs) (im Folgenden „OMNeT++ FSM Graphical Editor (OFGE)“ genannt) entwickelt und mit Hilfe von GMF realisiert (siehe Abschnitt 5.1).

2.3. Codegenerierung

Grundsätzlich ist es mit fast jeder Programmiersprache möglich, Code zu generieren. Um einen Code überhaupt generieren zu können, muss ein Modell vorhanden sein, auf dem die Codegenerierung basiert. Dieses Modell gilt als Input für die Codegenerierung und besteht meistens aus einer XML-Datei. Zusätzlich müssen statische Regeln festgelegt werden, die zusammen mit dem Eingabemodell die Artefakte generieren. Solche Artefakte können z.B. Programmcodes und Konfigurationen sein. Eine Programmiersprache, die für die Codegenerierung eingesetzt wird, sollte folgende Aspekte erfüllen [vgl. Stahl, Thomas u. a., 2007, S. 145]:

- Schreiben von Text in Dateien.
- Auslesen von Daten aus dem Modell.
- Zusammenführung von Modelldaten und satischem Text.

Für diese Arbeit wurde die Extensible Stylesheet Language Transformation (XSLT) [World Wide Web Consortium, 2007] Programmiersprache für die Codegenerierung ausgewählt (siehe Abschnitt 4.4). Dies ist eine Sprache, die nach den Regeln des XML-Standards aufgebaut ist und zur Transformation von XML-Code in beliebigen Text dient [vgl. Stahl, Thomas u. a., 2007, S. 147].

2.4. Finite State Machines

Finite State Machines (FSMs) bestehen aus Zuständen, Transitionen, Aktionen und Ereignissen. Ein Zustand beschreibt den Status eines Systems. Ereignisse treten zu bestimmten Zeitpunkten auf, diese führen zu einem Zustandswechsel. Ein Zustandswechsel ist vom Ereignis und dem aktuellen Zustand abhängig und wird mit Hilfe von Transitionen realisiert. Aktionen können sowohl in den Transitionen als auch in den Zuständen eingetragen werden [vgl. Balzert, 2005, S.87]. FSMs dienen dazu, das Verhalten von Systemen zu modellieren. Diese werden in mehreren Bereichen effektiv eingesetzt [vgl. Hopcroft, John u. a., 2011, S.25]:

- Verifizierung aller Arten von Systemen (z.B Kommunikationsprotokolle).
- Überprüfung des Verhaltens eines digitalen Schaltkreises.
- Durchsuchen von Texten (z.B. um bestimmte Ausdrücke zu finden).
- Modellierung von Systemen, Protokollen usw.

Als Beispiel hierfür wird in Abbildung 2.4 mit Hilfe der Unified Modeling Language (UML) [Object Management Group] eine FSM dargestellt. Die UML ist eine typische Notation zur Repräsentation von FSMs. Wie in Abbildung 2.4 zu sehen ist, besteht die FSM aus den Zuständen:

- *INIT*
- *SLEEP*
- *ACTIVE*

Der *INIT*-Zustand stellt den Startpunkt der FSM dar. In diesem Fall wäre dies der *SLEEP*-Zustand. Die Zustände werden über Transitionen miteinander verbunden. Transitionen sind gerichtete Kanten, die einen Zustandsübergang vom Quell- zum Zielzustand darstellen. Diese können durch boolesche Ausdrücke (engl. Guards) bedingt sein. Bedingte Transitionen dürfen erst ausgeführt werden, wenn der Guard zu *ture* ausgewertet wird. Beispielsweise wird in Abbildung 2.4 die FSM von *ACTIVE* zu *SLEEP* erst wechseln, wenn *msg==startStopBurst* zu *true* ausgewertet wird. Zusätzlich gibt es Entry- und

Exit-Aktionen. Entry-Aktionen werden beim Betreten eines Zustands ausgeführt. Exit-Aktionen werden beim Verlassen eines Zustands ausgeführt. Beispielsweise wird die Anweisung *if(msg==sendMessage) send(job, „out“);* beim Betreten von *AKTIVE* ausgeführt [vgl. Kecher, 2011, Abschnitt 10.3].

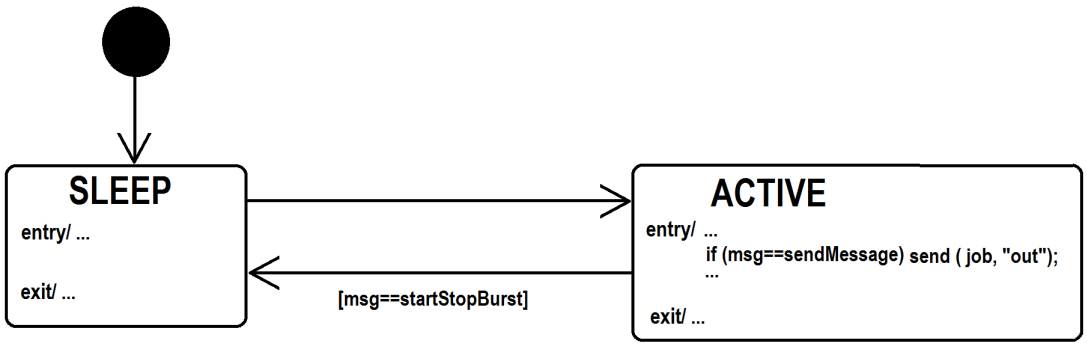

Abbildung 2.4.: FSM-Beispiel in der UML-Notation

2.4.1. Finite State Machines von OMNeT++

Die FSM, die OMNeT++ zur Verfügung stellt, wird in dieser Arbeit eingesetzt. Daraufhin werden die Grundlagen der OMNeT++ FSM erläutert.

Finite State Machines (FSMs) eignen sich gut für die Implementierung von Kommunikationsprotokollen. Daher bietet OMNeT++ eine Klasse und eine Menge von Makros, um FSMs aufzubauen.

Semantik der OMNeT++ FSM:

- Es gibt zwei Arten von Zuständen: Steady und Transient. Bei jedem Event (d.h. bei jedem Aufruf von *handleMessage()*) springt die FSM aus dem aktuellen Steady-Zustand, läuft durch eine Reihe von Transient-Zuständen und wechselt schließlich in einen anderen Steady-Zustand. Somit kann sich das System zwischen zwei Events nur in einem der Steady-Zustände befinden. Transient-Zustände sind daher nicht notwendig. Diese existieren, um die FSM zu strukturieren, indem die Aktionen eines Wechselvorgangs in den Transient-Zuständen gruppiert werden.
- Programmcode in einem Zustand kann an zwei Stellen zugewiesen werden, beim Eintreten bzw. Verlassen eines Zustandes (bekannt als Entry-/Exit-Code).
- Wenn die FSM nach dem Eintreten eines Events im selben Zustand bleiben soll, muss der Zustand verlassen und neu betretet werden.
- Der Entry-Code sollte den Zustand nicht ändern (dies wird von OMNeT++ überprüft), daher müssen Zustandsübergänge (engl. Transitions) in den Exit-Code stattfinden.

Die OMNeT++ FSMs können verschachtelt werden. Das bedeutet, dass jeder Zustand im Entry- bzw. Exit-Code eine weitere FSM beinhalten kann. So können Subzustände eingeführt werden, um damit eine gewisse Struktur in die FSM zu bringen, falls diese zu groß wird [vgl. Manual OMNeT++ Community, b, Abschnitt 4.10].

OMNeT++ FSM Implementierung

Die OMNeT++ FSM ist in einem Objekt vom Typ *cFSM* gespeichert. Die möglichen Zustände werden in einem Enum definiert. Zusätzlich wird in dem Enum jeweils der

Zustandstyp (Transient bzw. Steady) definiert. Im folgenden Beispiel hat die FSM die beiden Steady-Zustände *SLEEP* und *ACTIVE* und den Transient-Zustand *SEND*. Jedem Zustand wird eine ID zugewiesen, diese muss innerhalb eines Zustandstyp eindeutig sein (siehe Listing 2.3). Zusätzlich hat jede FSM einen Startzustand, dieser wird mit dem Wert 0 initialisiert und in der Regel *INIT* genannt [vgl. Manual OMNeT++ Community, b, Abschnitt 4.10].

```
enum {
        INIT = 0,
        //state name = state type(id)
        SLEEP = FSM_Steady(1),
        ACTIVE = FSM_Steady(2),
        SEND = FSM_Transient(1),
};
```

Listing 2.3: Beispiel der Definition von Zuständen der OMNeT++ FSM (Codeausschnitt aus dem Fifo2 Sample der OMNeT++ Simulation)

Die OMNeT++ FSM ist in einen *FSM_Switch()* eingebettet. Der *FSM_Switch()* entspricht einer Standard-*switch*-Anweisung [Prinz und Kirch, 2013, S. 106], die in einer *for*-Schleife eingebettet ist. Die *for*-Schleife wird so lange durchlaufen, bis die *switch*-Anweisung einen Steady-Zustand erreicht. Die Implementierung hierzu ist in der *cfsm.h* Library zu finden. In diesem Ansatz werden Endlosschleifen durch das Zählen von Zustandsübergängen vermieden. Wenn die FSM 64 Zustandsübergänge durchläuft, ohne einen Steady-Zustand zu erreichen, wird die Simulation mit einer Fehlermeldung beendet. Im *FSM_Switch()* hat jeder Zustand seinen Enter-/Exit-Case (siehe Listing 2.4) [vgl. Manual OMNeT++ Community, b, Abschnitt 4.10].

```
FSM_Switch(fsm)
{
        case FSM_Exit(state1):  //...
                                break;
        case FSM_Enter(state1): //...
                                break;
        case FSM_Exit(state2):  //...
                                break;
        case FSM_Enter(state2): //...
                                break;
};
```

Listing 2.4: *FSM_Switch()* Beispiel der OMNeT++ FSM

Zustandsübergänge (engl. Transitions) werden über den Aufruf *FSM_Goto()* realisiert, dieser speichert den neuen Zustand in dem *cFSM* Objekt:

FSM_Goto(fsmObj, newState);

Beispiel

Als Beispiel wird dieselbe FSM, die in Abbildung 2.4 dargestellt wurde, als OMNeT++ FSM dargestellt (siehe Abbildung 2.5). Diese wird ebenso mit Hilfe der UML-Notation dargestellt. Wie zu sehen ist, wird der Sendevorgang von *ACTIVE* als Transient-Zustand *SEND* dargestellt, um der FSM eine Struktur zu geben (siehe gestrichelter Zustand). Die Zustände *SLEEP* und *ACTIVE* sind vom Typ Steady.

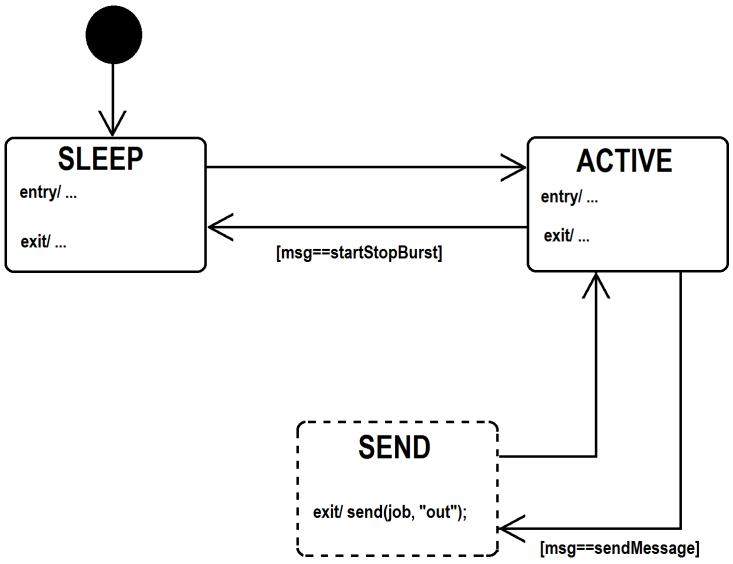

Abbildung 2.5.: Das Beispiel von Abbildung 2.4 als OMNeT++ FSM dargestellt

Die FSM in Abbildung 2.4 und Abbildung 2.5 basiert auf dem Fifo Sample der OMNeT++ Simulation. Der zugehörige Code wird in Listing 2.5 gezeigt. Im *SLEEP*-Zustand befindet sich das Modul in einem Ruhezustand. Im *ACTIVE*-Zustand sendet das Modul Nachrichten. Die Kommentare im Listing 2.5 verweisen auf die einzelnen Elemente der FSM

(Transitionen, Enter-/Exit-Cases usw.) [vgl. Manual OMNeT++ Community, b, Abschnitt 4.10].

```
#define FSM_DEBUG
#include <omnetpp.h>

class BurstyGenerator : public cSimpleModule
{    protected:
        //Parameter
        double sleepTimeMean;
        double burstTimeMean;
        double sendIATime;
        cPar *msgLength;
        //Die FSM und deren Zustaende
        cFSM fsm;
        enum {                                  //Definition der Zustaende
            INIT = 0,
            SLEEP = FSM_Steady(1),              //SLEEP = Steady
            ACTIVE = FSM_Steady(2),             //ACTIVE = Steady
            SEND = FSM_Transient(1),            //SEND = Transient
        };

        //Verwendete Variablen
        int i;
        cMessage *startStopBurst;
        cMessage *sendMessage;

        //Virtuelle Funktionen
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
};

Define_Module(BurstyGenerator);

void BurstyGenerator::initialize()
{
    fsm.setName("fsm");
    sleepTimeMean = par("sleepTimeMean");
    burstTimeMean = par("burstTimeMean");
    sendIATime = par("sendIATime");
    msgLength = &par("msgLength");
    i = 0;
    WATCH(i);
    startStopBurst = new cMessage("startStopBurst");
    sendMessage = new cMessage("sendMessage");
    scheduleAt(0.0,startStopBurst);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
    FSM_Switch(fsm)             //Die FSM_Switch mit den einzelnen Zustaenden als Cases
    {
        case FSM_Exit(INIT):
            FSM_Goto(fsm,SLEEP);        //Transition ohne Bedingung von INIT zu SLEEP
            break;
        case FSM_Enter(SLEEP):          //Enter Case von SLEEP
            //Schedule den nächsten Burst-Anfang
            scheduleAt(simTime()+exponential(sleepTimeMean),
```

```
56              startStopBurst);
57              break;
58          case FSM_Exit(SLEEP):           //Exit Case von SLEEP
59              //Schedule den aktuellen Burst-Ende
60              scheduleAt(simTime()+exponential(burstTimeMean),
61              startStopBurst);
62              //Transition zu ACTIVE Zustand
63              if (msg!=startStopBurst) {
64                 error("invalid_event_in_state_ACTIVE");
65              }
66              FSM_Goto(fsm,ACTIVE);       //Transition ohne Bedingung von SLEEP zu ACTIVE
67              break;
68          case FSM_Enter(ACTIVE):         //Enter Case von ACTIVE
69              //Schedule den nächsten Sendevorgang
70              scheduleAt(simTime()+exponential(sendIATime), sendMessage);
71              break;
72          case FSM_Exit(ACTIVE):          //Exit Case von ACTIVE
73
74              if (msg==sendMessage) {
75                  FSM_Goto(fsm,SEND);     //Bedingte Transition von ACTIVE zu SEND
76              } else
77              if (msg==startStopBurst) {
78                  cancelEvent(sendMessage);
79                  FSM_Goto(fsm,SLEEP);    //Bedingte Transition von ACTIVE zu SLEEP
80              } else {
81                  error("invalid_event_in_state_ACTIVE");
82              }
83              break;
84          case FSM_Exit(SEND):            //Exit Case von SEND
85              {
86              //Job generieren und senden
87              char msgname[32];
88              sprintf( msgname, "job-%d", ++i);
89              ev << "Generating_" << msgname << endl;
90              cMessage *job = new cMessage(msgname);
91              job->setBitLength( (long) *msgLength );
92              job->setTimestamp();
93              send( job, "out" );         //Sendevorgang
94              //Transition zu ACTIVE Zustand
95              FSM_Goto(fsm,ACTIVE);       //Transition ohne Bedingung von SLEEP zu ACTIVE
96              break;
97          }
98      }
99  }
```

Listing 2.5: „BurstyGenerator“ Modul aus dem Fifo2 Sample der OMNeT++ Simulation

OMNeT++ FSM Debugging

Die OMNeT++ FSM kann ihre Zustandsübergänge ausgeben (siehe Listing 2.6). Hierzu gibt es ein vordefiniertes Makro, das vor dem *#include <omnetpp.h>* definiert werden muss:

> ***#define** FSM_DEBUG //enables debug output from FSMs*
> ***#include** <omnetpp.h>*

```
...
FSM GenState: leaving state SLEEP
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SEND
FSM GenState: leaving state SEND
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SLEEP
...
```

Listing 2.6: Ausgabe der Zustandsübergänge der OMNeT++ FSM

Die Ausgabe wird mit Hilfe des Makros *FSM_Print()* abgewickelt (siehe Listing 2.7). Das Standard-Ausgabeformat kann geändert werden, indem der *FSM_Print()* nach dem *#include <omnetpp.h>* neu definiert wird [vgl. Manual OMNeT++ Community, b, Abschnitt 4.10].

```
#define FSM_Print(fsm,exiting)
        (ev << "FSM_" << (fsm).getName()
                << ((exiting) ? ":_leaving_state_" : ":_entering_state_")
                << (fsm).getStateName() << endl)
```

Listing 2.7: Vordefiniertes *FSM_Print()* Makro der *cfsm.h* Library

3. Anforderungen

Im Mittelpunkt dieser Arbeit stehen die Entwicklung und Implementierung eines graphischen Editors zur Modellierung von FSMs. In diesem Kapitel werden die Anforderungen an die verschiedenen Ebenen dieser Arbeit definiert.

3.1. Anforderungen an das FSM-Entwurfsmuster

Die FSMs, die mit Hilfe des graphischen Editors modellierbar sind, sollen nach dem Unified Modeling Language (UML) Standard [Object Management Group] repräsentiert werden, da diese eine typische Notation zur Repräsentation von FSMs ist. Dabei sollen FSMs nach einem bestimmten Entwurfsmuster implementiert werden, das dazu dient, FSMs für Protokolle zu realisieren. Da die FSMs nach dem UML-Standard repräsentiert werden, gelten nach Scheibl folgende Anforderungen an das FSM-Entwurfsmuster [vgl. Scheibl, Kapitel 13]:

- Die Implementierung der FSMs soll in der C++ Sprache erfolgen.
- Zustände können unterschiedliche Typen haben.
- Zustände sollen unterscheidbar sein.
- Zustände sollen Entry- und Exit-Aktionen haben können.
- Transitionen sollen auf Ereignisse reagieren können und somit den Zustand ändern.
- Ein Wächterausdruck (engl. guard) soll eine Transition schützen können.
- Es soll möglich sein, einer Transition Aktionen (engl. effect) zuzuordnen.
- Die FSMs sollen auf folgende Arten von Ereignissen reagieren können:

  - Signale
  - Methodenaufrufe
  - Wechsel eines Wertes
  - Zeit (z.B. Timer)

3.2. Anforderungen an den graphischen Editor

Der Nutzer soll in der Lage sein, die FSM, die das Verhalten einer OMNeT++ Komponenten beschreibt, mit Hilfe eines graphischen Editors modellieren zu können. Für den graphischen Editor gelten folgende Anforderungen:

- Er soll für die Verwendung als Eclipse-Plugin entwickelt werden, um ihn in OMNeT++ zu integrieren.
- Er soll das graphische Bearbeiten der FSM (Zustände und deren Aktionen sowie die Transitionen usw.) ermöglichen.
- Er soll eine Palette zur Verfügung stellen, welche die verfügbaren Werkzeuge zur Modellierung einer FSM bereitstellt (Zustand, Transition usw.).
- Er soll die Daten der modellierten FSM (wie z.B. welche Zustände vorhanden sind und deren Position, über welche Transitionen diese verbunden sind usw.) in einem Dateiformat (z.B. XML) hinterlegen. Diese Daten sind erforderlich, um die FSMs im Editor wieder einlesen zu können und den Implementierungscode für die modellierten FSMs zu generieren (siehe Abschnitt 3.3).
- Er soll in der Lage sein, folgende graphische Repräsentationen für die FSM-Komponenten bereitzustellen:
  - **Initialzustand:** Ein schwarz gefüllter Kreis.
  - **Zustand:** Ein Rechteck mit runden Ecken und ein zugehöriges Label, in dem der Zustandsname eingetragen wird.
  - **Transition:** Eine gerichtete Kante und ein Label, in dem Guard und Effect eingetragen werden.
  - **Entry-Aktionen:** Ein Label, in dem der Entry-Code eingetragen wird.

  - **Exit-Aktionen:** Ein Label, in dem der Exit-Code eingetragen wird.

- Er soll in der Lage sein, feste Regeln zu definieren und zu überprüfen, die bei der Modellierung einer FSM sichergestellt sind:
  - Ein Initial-Zustand kann nur ausgehende Transitionen haben.
  - Ein Zustand kann ein- und ausgehende Transitionen haben.
  - Eine Transiton kann entweder zwei Zustände oder einen Zustand mit sich selber verbinden.
  - Enty- und Exit-Labels können nur innerhalb eines Zustandes eingetragen werden.

3.3. Anforderungen an die Codergenerierung

Wie im vorigen Abschnitt 3.2 erwähnt wurde, soll der graphische Editor für jede modellierte FSM ein Dateiformat bereitstellen. Dieses enthält Daten, die die graphische Darstellung der FSM beschreibt und für das Einlesen der FSM im Editor nötig sind. Außerdem muss das Format dieser Daten für die Codegenerierung lesbar sein. Die Daten müssen automatisch eingelesen werden, um folgendes zu generieren:

- Quellcode, der der in der C++ Sprache modellierten FSM entspricht und somit das Verhalten der OMNeT++ Komponente dieser FSM beschreibt.
- Quellcode, der der graphischen Darstellung der FSM in der OMNeT++ Simulation zur Laufzeit entspricht (siehe Abschnitt 3.4).

Somit muss die Programmiersprache, welche für die Codegenerierung angewendet wird, in der Lage sein, Text mit bestimmten Formaten aus einer Datei auszulesen, diesen in ein anderes Format (z.B. Java, OMNeT++, NED usw.) umzuwandeln und ihn in eine oder mehrere Dateien zu schreiben.

3.4. Anforderungen an die graphische Darstellung der FSM

Nachdem der Nutzer die FSM modelliert hat, der Implementierungscode generiert wurde und die OMNeT++ Simulation gestartet ist, muss dieser in der Lage sein, die FSM zur Laufzeit zu beobachten. Somit kann der Nutzer zu einem bestimmten Zeitpunkt wissen, in welchem Zustand sich eine bestimmte Komponente befindet. Um dies zu ermöglichen, muss die FSM zur Laufzeit graphisch dargestellt werden. Für die graphische Darstellung gelten folgende Anforderungen:

- Die FSM, die zur Laufzeit dargestellt wird, und die FSM, die mit Hilfe des Editors modelliert wurde, sollen gleich aussehen.
- Die graphisch dargestellte FSM muss sich zur Laufzeit aktualisieren können (z.B. soll der aktuelle Zustand rot markiert werden).
- Die graphisch dargestellte FSM soll in der graphischen Schnittstelle der OMNeT++ Simulation (Tkenv) [vgl. Manual OMNeT++ Community, b, Abschnitt 10.3] integriert werden.

4. Konzept

Dieses Kapitel beschreibt zunächst das Konzept des gesamten Aufbaus dieser Arbeit.

4.1. Architekturkonzept

Wie in Abbildung 4.1 zu sehen ist, besteht der Aufbau dieser Arbeit aus mehreren Schichten, die aufeinander aufbauen. In der ersten Schicht befindet sich der OMNeT++ FSM Graphical Editor (OFGE), der mittels Graphical Modeling Framework (GMF) [Eclipse Foundation, d] implementiert und entwickelt wurde. Mit diesem steht dem Nutzer ein Werkzeug zur Verfügung, um gewünschte OMNeT++ FSMs graphisch zu modellieren. Der OFGE bietet für jede modellierte FSM eine XML-Datei an (OFGE-Output). Diese enthält die Informationen zur graphischen Darstellung der modellierten FSM (z.B. welche Zustände vorhanden sind und jeweils die Position, welche Transitionen es gibt usw.). Diese werden für das Einlesen der FSM im OFGE benötigt (OFGE-Input). Zusätzlich sind die XML-Daten für die Codegenerierung nötig und werden somit an die zweite Schicht übergeben (XSLT-Input) (siehe Abbildung 4.1).
Die zweite Schicht entspricht der XSL-Transformation [World Wide Web Consortium, 2007], mit der die Codegenerierung abgewickelt wird. Die Codegenerierung erzeugt folgende Dateien (XSLT-Output):

1. Eine **.cc*-Datei:
   - entspricht dem OMNeT++ Modul, für das die FSM mittels OFGE modelliert wurde.
   - beschreibt die OMNeT++ FSM in der C++ Programmiersprache.
2. Eine **_fsm_updater.cc*-Datei:
   - entspricht ebenso einem OMNeT++ Modul.

- hat die Aufgabe, die modellierte FSM zur Laufzeit in der OMNeT++ Simulation darzustellen und zu aktualisieren.

3. Eine **.ned*-Datei:
   - Diese beschreibt den Aufbau der oben genannten Module in der NED-Sprache.

Die drei aufgezählten erzeugten Dateien werden in den Abschnitten 5.2.2, 5.2.3 und 5.2.4 ausführlicher vorgestellt.

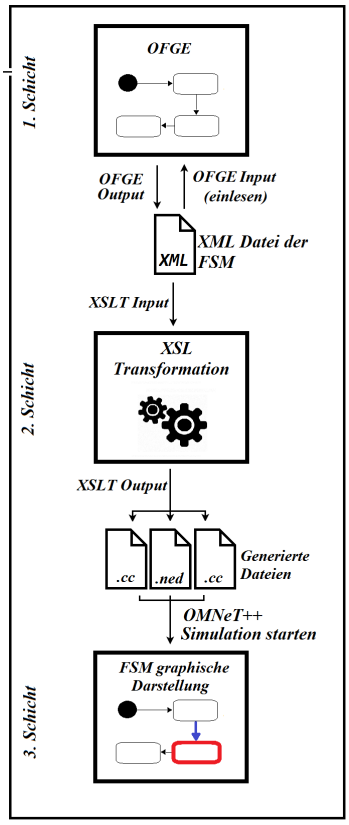

Abbildung 4.1.: Die einzelnen Schichten zum Aufbau dieser Arbeit

In den folgenden Abschnitten wird die Konzeption für jede einzelne Schicht erläutert.

4.2. FSM-Entwurfsmuster

Ziel dieser Arbeit ist es, FSMs in die OMNeT++ Simulation zu integrieren. Dafür gibt es eine Menge von Mechanismen bzw. Pattern, die zur Auswahl stehen, um FSMs zur Beschreibung von Protokollen in C++ schreiben zu können. Zwei dieser Implementierungsansätze werden in diesem Abschnitt diskutiert: das State Pattern [Hauer, 2009–2010] und der OMNeT++ FSM Mechanismus.

4.2.1. State Pattern

Das State Pattern ist ein Entwurfsmuster, das in der Softwareentwicklung zu der Kategorie Verhaltensmuster gehört. Im Falle eines Objekts, dessen Verhalten zur Laufzeit in Abhängigkeit von seinem Zustand geändert werden muss, wird dieses eingesetzt. Für jeden Zustand wird eine Klasse definiert. Dadurch können die einzelnen Zustände als unabhängige Objekte behandelt werden [vgl. Gamma, Erich u. a., 2015, S. 374].

Beispiel

In der Bachelorarbeit von Lazar Todorov Todorov [Todorov Todorov, 2013] wurde ein Synchronisationsprotokoll (AS6802) implementiert. Die FSMs des Synchronisationsprotokolls wurden nach dem State Pattern implementiert. In dem Synchronisationsprotokoll können die Netzwerkkomponenten drei Rollen einnehmen:

- Synchronisation Client (SC)
- Synchronisation Master (SM)
- Compression Master (CM)

Jede dieser Netzwerkkomponenten kann ein End-System bzw. ein Switch sein. Diese werden als SC, SM oder CM konfiguriert. Dabei hat jede Komponente eine Menge von Zuständen. Als Beispiel wird der SC verwendet. Dieser kann sich zur Laufzeit in mehreren Zustände befinden:

- *INIT*
- *INTEGRATE*

- *STABLE*
- *PSEUDOSYNC*

Bei der Implementierung der SC FSM nach dem State Pattern wird ein Kontext definiert (siehe *StateContex* in Abbildung 4.2). Dieser entspricht der Schnittstelle zur Außenwelt und hat die Aufgabe, die Zustandsklassen zu verwalten. Die Klasse *SC_State* definiert eine Schnittstelle zur Kapselung der Verhaltensweise. Die Unterklassen implementieren die unterschiedlichen Verhaltensweisen und definieren somit die einzelnen Zustände des SC [vgl. Gamma, Erich u. a., 2015, S.374-375].

Vorteile

- Erweiterbarkeit und Änderungsstabilität: Da die Zustände unabhängige Objekte (Klassen) sind, können diese einfach erweitert bzw. geändert werden.
- Intuitivität und Verständlichkeit: Durch die Kapselung der Zustände in mehreren Klassen ist der Code einfach verständlich..
- Explizite Zustandsübergänge: Durch die Einführung von Objekten für jeden Zustand wird der Zustandswechsel explizit.
- Die Wartbarkeit wird erhöht und Zustandsobjekte können wiederverwendet werden.
- Zustände können unterschiedliche Typen haben.
- Entry- und Exit-Aktionen werden unterstützt.
- Transitionen können durch Wächterausdrücke geschützt werden.
- Aktionen können Transitionen zugeordnet werden [Hauer, 2009–2010].

Nachteile

- Erhöhte Klassenanzahl.
- Weniger kompakt als eine einzige Klasse [Hauer, 2009–2010].

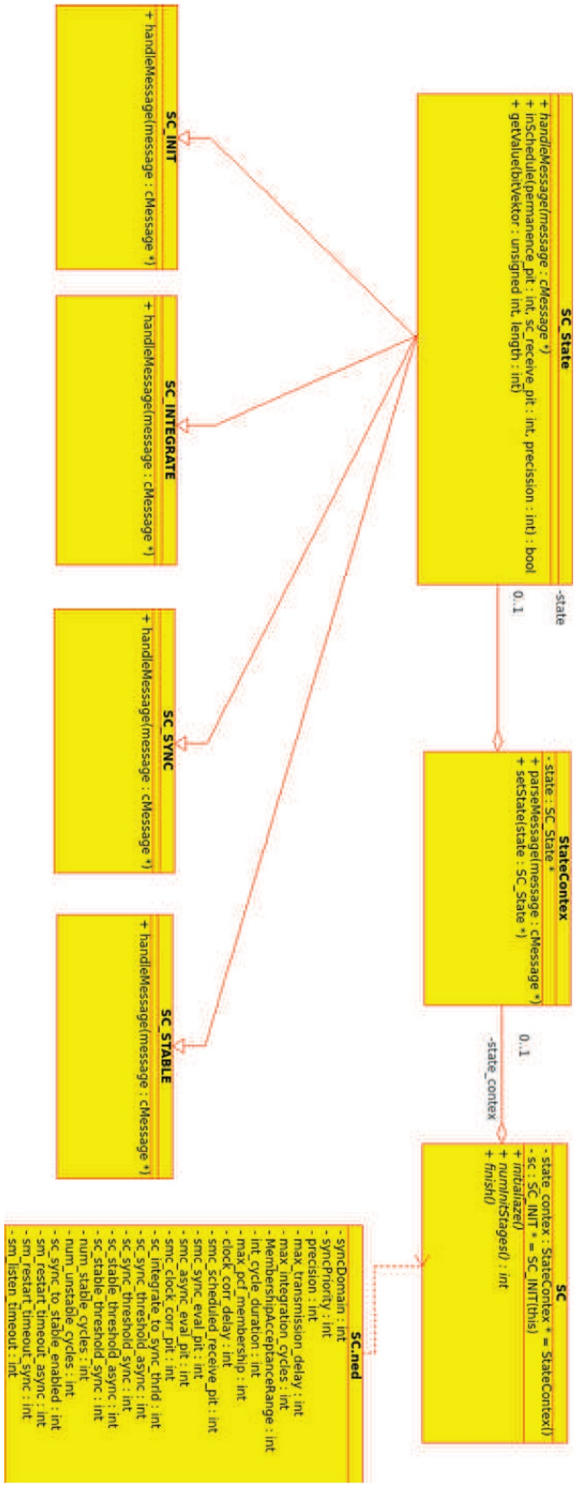

Abbildung 4.2.: SC - UML Diagramm (nach dem State Pattern implementiert) [Todorov Todorov, 2013]

4.2.2. OMNeT++ FSM

Die OMNeT++ Simulationsumgebung bietet einen grundlegenden Mechanismus zur Implementierung von FSMs. Dabei stellt diese eine API bereit, um die Programmierung der FSMs für den Entwickler zu vereinfachen. Die API ist mit Hilfe einer Reihe von Makros realisiert und dient zur Strukturierung des Codes (mehr zu den Grundlagen der OMNeT++ FSM in Abschnitt 2.4.1).

Beispiel

Als Beispiel wird der SC aus dem vorigen Abschnitt 4.2.1 verwendet. Wie schon erwähnt hat dieser folgende Zustände: *INIT*, *INTEGRATE*, *STABLE* und *PSEUDOSYNC*. Um die FSM des SC nach dem OMNeT++ Mechanismus zu implementieren, werden keine zusätzlichen Klassen gebraucht. Die Implementierung findet in dem SC-Modul statt. In der *handleMessage()* Funktion des SC-Moduls wird mit Hilfe des *FSM_Switch()* (siehe Abschnitt 2.4.1) die SC FSM definiert. Jeder Zustand entspricht dabei einem Entry- und Exit-Case der *FSM_Switch()* (siehe Abbildung 4.3).

Abbildung 4.3.: SC - UML Diagramm (nach dem OMNeT++ Mechanismus implementiert)

Vorteile

- Die OMNeT++ FSM ist dem OMNeT++ Nutzer bekannt. Dementsprechend muss er sich nicht mit neuen FSMs auseinandersetzen.
- Zusätzliche Klassen, um die Zustände zu definieren, sind nicht erforderlich.
- Zustände können unterschiedliche Typen haben (Steady, Transient).
- Durch die Transient-Zustände, die die OMNeT++ FSM anbietet, kann eine Menge von Aktionen aufgeteilt werden. Somit können diese in übersichtlicher, verständlicher Form abgehandelt werden.
- Verständlich und kompakt.
- Vereinfachte und strukturierte Programmierung durch die bereitgestellten Makros.
- Das Entwurfsmuster ist ein Bestandteil der OMNeT++ Simulationsumgebung und somit in C++ implementierbar.
- Transitionen können durch Wächterausdrücke geschützt werden.
- Aktionen können Trasnitionen zugeordnet werden.

Nachteile

- Unübersichtlichkeit bei großen FSMs, da jeder Zustand einem Entry- und einem Exit-Case entspricht.
- Schlechte Wartbarkeit, da alle Zustände in einem Objekt definiert sind und somit voneinander abhängig sind.
- Schlechte Wiederverwendbarkeit. Da die Zustände nicht als unabhängige Objekte implementiert werden, können diese nur von einem Modul genutzt werden, nicht von mehreren.

4.2.3. Auswahl des Entwurfsmusters

Ziel dieser Arbeit ist es, mit Hilfe eines Editors die FSMs modellieren zu können. Der dazugehörige Code soll generiert werden. Beide der oben genannten Ansätze (State Pattern und OMNeT++ FSM Mechanismus) erfüllen die Anforderungen, die an das Entwurfsmuster gestellt wurden (siehe Abschnitt 3.1). Durch die Codegenerierung sind die Vorteile (Wartbarkeit und Wiederverwendbarkeit) des State Patterns nicht mehr relevant. Ein weiterer Nachteil des State Patterns ist die hohe Anzahl an Klassen, die für die Implementierung nötig ist. Beim OMNeT++ FSM Entwurfsmuster wird die Anzahl der Klassen deutlich verringert (siehe Abbildung 4.2 im Vergleich zu Abbildung 4.3). Aus diesen Gründen wurde bei der Wahl des Entwurfsmusters entschieden, das OMNeT++ FSM Entwurfsmuster einzusetzen.

4.3. GMF graphischer Editor

Ziel dieser Arbeit ist es, dem Nutzer zu ermöglichen, die FSMs mit Hilfe eines grafischen Editors zu modellieren, sodass der dazugehörige Code durch die modellierte FSM generiert wird (siehe erste Schicht in Abbildung 4.1 unter dem Abschnitt 4.1). Mit Hilfe des Graphical Modeling Frameworks (GMF) [Eclipse Foundation, d] wird der OMNeT++ FSM Graphical Editor (OFGE) entwickelt, der zur Modellierung von FSMs dient. Dementsprechend befasst sich dieser Abschnitt mit der Konzeptionierung von GMF.

4.3.1. Einleitung

Graphical Modeling Framework (GMF) ist ein Projekt der Eclipse-Open-Source-Gemeinschaft. Es ermöglicht die Entwicklung von grafischen Editoren auf Basis von Eclipse Modeling Framework (EMF) [Eclipse Foundation, b] und Graphical Editing Framework (GEF) [Eclipse Foundation, c] [vgl. Pietrek, Georg u. a., 2007, S. 75].

„GMF definiert ein Framework, das die Editierung von EMF-Modellen mit GEF-Editoren ermöglicht. Weiterhin werden eigene Domain Specific Languages (DSLs) zur Modellierung des Editors bereitgestellt. Aus diesen Modellen wird dann mit Hilfe des GMF-Codegenerators der Code für das Eclipse Editor Plug-in generiert.“ [Pietrek, Georg u. a., 2007, S. 75]

4.3.2. Schritte zum Aufbau

Um einen graphischen Editor mit Hilfe von GMF zu erstellen, sind die folgenden Schritte durchzuführen [vgl. Pietrek, Georg u. a., 2007, S. 75-76]:

- Definition des Domain Models (auch Metamodell[1] genannt). Dieses lässt sich mit Hilfe von EMF bearbeiten und gilt als Basis für die Erzeugung des Editors. Das Domain Model definiert alle Domänenelemente im Editor, die dem Nutzer für die Modellierung der GUI (in diesem Fall die OMNeT++ FSM) bereitgestellt werden sollen.
- Das Domain Generator Model (**.genmodel*) aus dem Domain Model ableiten. Durch das Domain Generator Model wird der Code für das Domain Model mit Hilfe von EMF generiert (siehe Abbildung 4.4).
- Das Graphical Definition Model (**.gmfgraph*) aus dem Domain Model ableiten und anschließend nach Wunsch bearbeiten. In diesem werden die visuellen Eigenschaften der Notationselemente beschrieben (z.B. Form, Hintergrundfarbe usw.) (siehe Abbildung 4.4).
- Das Tool Definition Model (**.gmftool*) aus dem Domain Model ableiten und anschließend nach Wunsch bearbeiten. In diesem wird u. a. die Tool-Palette beschrieben. Diese enthält die Werkzeugelemente, mit welchen die OMNeT++ FSM modelliert wird (siehe Abbildung 4.4).
- Verknüpfung der drei vorigen Modelle (Domain Model, Graphical Definition Model und Tool Definition Model) durch ein Mapping Model (**.gmfmap*). Somit wird festgelegt, welche Domänenelemente auf welchen Notations- und Werkzeugelementen abgebildet werden sollen (siehe Abbildung 4.4).
- Das GMF Generator Model (**.gmfgen*) durch eine Transformation aus dem Mapping Model ableiten (siehe Abbildung 4.4).
- Mit Hilfe des GMF Generator Models und des vom Domain Generator Model generierten Codes wird der GMF graphische Editor erzeugt (im Folgenden „OMNeT++ FSM Graphical Editor (OFGE)“ genannt) (siehe Abbildung 4.4).

[1] *„Ein Metamodell ist ein Modell, das eine Menge anderer Modelle definiert, die als Instanzen des Metamodells bezeichnet werden“* [Reussner und Hasselbring, 2006, S. 106]

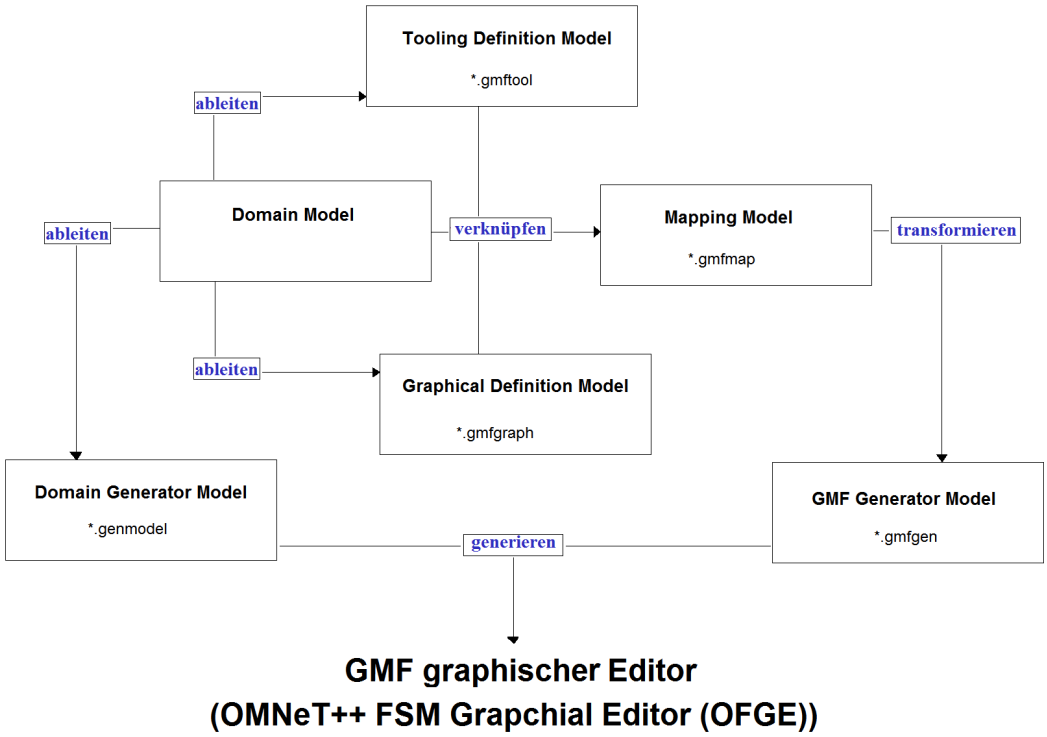

Abbildung 4.4.: Schritte zum Aufbau des OFGE mittels GMF

GMF bietet für jedes Diagramm, das mit Hilfe des erzeugten Editors modelliert wurde, eine XML-Datei an. Dementsprechend bietet OFGE für jede modellierte FSM eine XML-Datei an. Diese enthält Informationen für die graphische Darstellung der modellierten FSM (z.B. vorhandene Zustände und jeweils die Position, Transitionen usw.).

4.4. Codegenerierung mittels XSLT

Wie in Abbildung 4.1 unter dem Abschnitt 4.1 zu sehen ist, wird in der zweiten Schicht ein Verfahren benötigt, um die Codegenerierung durchzuführen. Aus der XML-Datei, die der GMF Editor bereitstellt, soll eine Menge von Dateien generiert werden (siehe Abbildung 4.1). Für diesen Zweck wird die XSL-Transformation [World Wide Web Consortium, 2007], kurz XSLT, eingesetzt. Dementsprechend wird in diesem Abschnitt die Konzeptionierung der XSL-Transformation erklärt.

4.4.1. XSL-Transformation

„Die Abkürzung für XSLT steht für eXtendable Stylesheet Language Transformation. XSLT gehört zu den deklarativen Sprachen und beschreibt und steuert die Umwandlung (Transformation) von XML-Dokumenten in andere Formate wie HTML, XHTML, Text oder andere XML-Strukturen.“ [Bongers, 2004, S. 26]

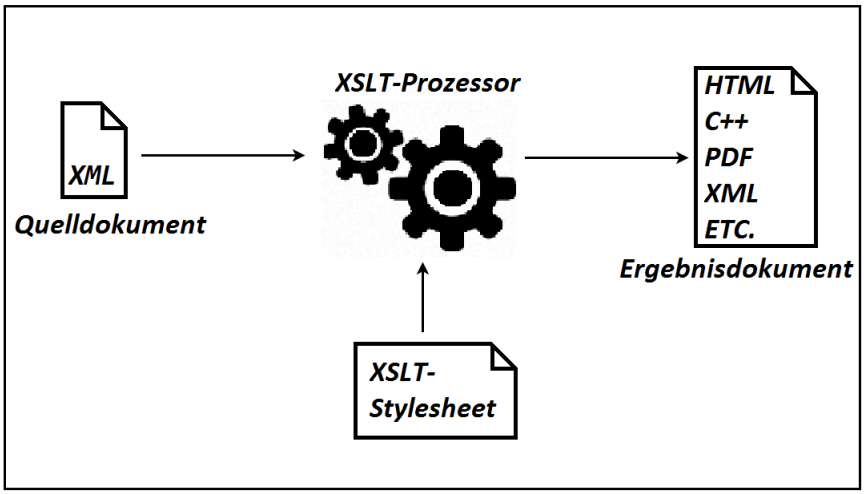

Abbildung 4.5.: Schematische Darstellung einer XSL-Transformation

Wie in Abbildung 4.5 zu sehen ist, spielen bei der XSL-Transformation drei Dateien eine wichtige Rolle [vgl. Bongers, 2004, S. 27-28]:

1. **Die Quelldatei:**
   Diese entspricht einer XML-Datei und enthält alle Informationen der FSM, die mit Hilfe vom OFGE modelliert wurde.

2. **Das XSLT-Stylesheet:**
   In diesem werden die Übersetzungsregeln definiert. Das Stylesheet wird vom XSLT-Prozessor eingelesen und anhand der Übersetzungsregeln wird aus der Quelldatei das gewünschte Ausgabeformat in der Zieldatei erzeugt.

3. **Die Zieldatei:**
   Diese lässt sich aus der Quelldatei und dem XSLT-Stylesheet erzeugen.

4.4.2. XSLT-Elemente

Wie im vorigen Abschnitt 4.4 schon erwähnt, beinhaltet das XSLT-Stylesheet eine Menge von Übersetzungsregeln. Diese werden in der Stylesheet-Hierarchie nach drei unterschiedlichen Kategorien geordnet [vgl. Bongers, 2004, S. 32]:

1. **Root-Elemente:**
   Die einzigen zwei XSLT-Elemente, die als Wurzelelemente einer XSLT-Datei eingesetzt werden können, sind *xsl:stylesheet* oder das Synonym *xsl:transform*. Ein Wurzelelement enthält alle anderen XSLT-Elemente.

2. **Toplevel-Elemente:**
   Direkte Kinderelemente des Wurzelelementes werden Toplevel-Elemente genannt. Diese definieren den globalen Aufbau des Stylesheets (wie z.B. Funktionsköpfe, Schablonen (Templates), Formatierungsregeln usw.).

3. **Instruktionen:**
   Diese Elemente sind den Toplevel-Elementen untergeordnet (Schleifen, Bedingungen, Templatesaufrufe usw.).

Die ausführliche Beschreibung der XSLT-Funktionsweise wird in Abschnitt 5.2 explizit erklärt. Es wird gezeigt, wie anhand der oben genannten XSLT-Elemente aus einer XML-Datei der C++ Code generiert wird.

4.5. Darstellung der OMNeT++ FSM

Ein Ziel dieser Arbeit ist es, dem Nutzer die Möglichkeit zu geben, die FSM, die mittels OFGE modelliert wurde, zur Laufzeit zu beobachten. Dies wird ermöglicht, indem die FSM in der OMNeT++ Simulation graphisch dargestellt wird. Die graphische Darstellung der FSM wird mit Hilfe der Canvas API, die OMNeT++ bereitstellt, in der OMNeT++ Simulation integriert.

4.5.1. Canvas API

Die Canvas API ist eines der Haupt-Highlights des OMNeT++ 5.0b1 Releases. Mit dieser ist es möglich, die grafische Benutzeroberfläche (Tkenv) der OMNeT++ Simulation mit graphischen Elementen (Figuren) zu erweitern. Solche graphischen Elemente sind beispielsweise Linien, Bilder, Text, Labels, Formen usw. [vgl. Varga, 2014].

4.5.2. Funktionsweise

In diesem Ansatz gibt es zwei grundlegende Komponenten:

- Canvases
- Figuren

Jedes Compound Modul kann einen Canvas zugewiesen werden. Dem Canvas können Figuren und weitere Canvases zugeordnet werden. Die Implementierung erfolgt über zwei Wege:

- Über die **.ned*-Datei: Die Figuren werden über das Property[1] @figure erzeugt. *@figure* hat eine Menge von Parametern z.B. Figurtyp, Position, Farbe usw., die übergeben werden können (siehe Listing 4.1).

[1] Im Allgemeinen werden @-Wörter, wie z.B. @figure, in NED Properties genannt. Diese werden verwendet, um verschiedene Objekte mit Metadaten zu annotieren. Properties können Dateien, Module, Parameter, Gates, Connections usw. zugewiesen werden [vgl. Manual OMNeT++ Community, b, Abschnitt 3.12].

```
module m {
        //create a rectangle with given position, size and color
        @figure[f](type=rectangle; coords=10,50; size=200,100; fillColor=green);
        //create the text "Hello World" inside of the rectangle "f"
        @figure[f.t](type=text; coords=20,80; text=Hello World);
}
```

Listing 4.1: Beispiel: Figuren der Canvas API über die NED-Datei erzeugen

- Über C++ Code in der **.cc*-Datei: Die Figuren sind normale Objekte, die zum Canvas-Objekt hinzugefügt werden (siehe Listing 4.2).

```
        ..
        cCancas *canvas = getCanvas();
        //creating rectangle figure
        cRectangleFigure *f = new cRectangleFigure();
        //setting color of rectangle
        f->setFillColor(cFigure::GREEN);
        //adding rectangle to canvas
        canvas->addFigure(f);
```

Listing 4.2: Beispiel: Figuren der Canvas API in C++ erzeugen

Zusätzlich können in C++ Figuren, die in NED definiert wurden, angesprochen werden, um diese zur Laufzeit manipulieren zu können (z.B. Farbe, Position, Orientierung einer Figur ändern) [vgl. Varga, 2014]

In dieser Arbeit wird die FSM eines OMNeT++ Moduls durch Canvas-Figuren zur Laufzeit graphisch dargestellt und aktualisiert (siehe Implementierung in den Abschnitten 5.2.3 und 5.2.4).

5. Implementierung

Bei der Implementierung wird die Erstellung des Plugins beschrieben. Zu Beginn wird gezeigt, wie mit Hilfe von GMF der OMNeT++ FSM Graphical Editor (OFGE) zur Modellierung von FSMs entwickelt wurde (siehe Abschnitt 5.1). Darauffolgend wird erklärt, wie anhand einer XML-Datei, die der OFGE bereitstellt, Implementierungscode mittels Extensible Stylesheet Language Transformation (XSLT) generiert wird (siehe Abschnitt 5.2). Anschließend wird die Zusammenführung von OFGE und XSLT beschrieben (siehe Abschnitt 5.3). Im Anhang dieser Arbeit befindet sich für das erzeugte Plugin ein englischsprachiges Manual für den Nutzer. Dieses enthält alle relevanten Informationen zur Bedienung und Installation des entwickelten Plugins (siehe Anhang).

5.1. OMNeT++ FSM Graphical Editor (OFGE)

In diesem Abschnitt wird beschrieben, wie der OMNeT++ FSM Graphical Editor (OFGE) zur Modellierung von FSMs mittels GMF (siehe Abschnitt 4.3) entwickelt und implementiert wurde.

Wie in Abschnitt 4.3.2 beschrieben, benötigt GMF sechs Dateien (Modelle), um einen graphischen Editor für ein bestimmtes Metamodell[1] zu generieren. Diese befinden sich in dem GMF Projekt *org.omnetpp.core.fsm.gmf* unter dem Ordner *model* (siehe Abbildung 5.1) und werden in den folgenden Abschnitten ausführlich beschrieben. GMF bietet ein Hilfswerkzeug an, *GMF Dashboard* genannt (*Window>Show View>Other>GMF Dashboard*). Dies dient als einfache Möglichkeit zur Erzeugung von graphischen Editoren. Mit Hilfe des *GMF Dashboards* können die einzelnen Modelle selektiert, editiert, abgeleitet bzw. transformiert werden (siehe Abbildung 5.2).

[1] *„Ein Metamodell ist ein Modell, das eine Menge anderer Modelle definiert, die als Instanzen des Metamodells bezeichnet werden“* [Reussner und Hasselbring, 2006, S. 106]

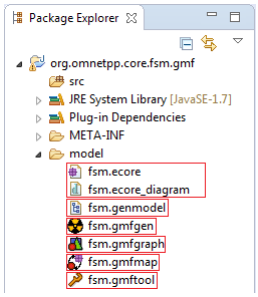

Abbildung 5.1.: Aufbau des GMF-Projekts *org.omnetpp.core.fsm.gmf*

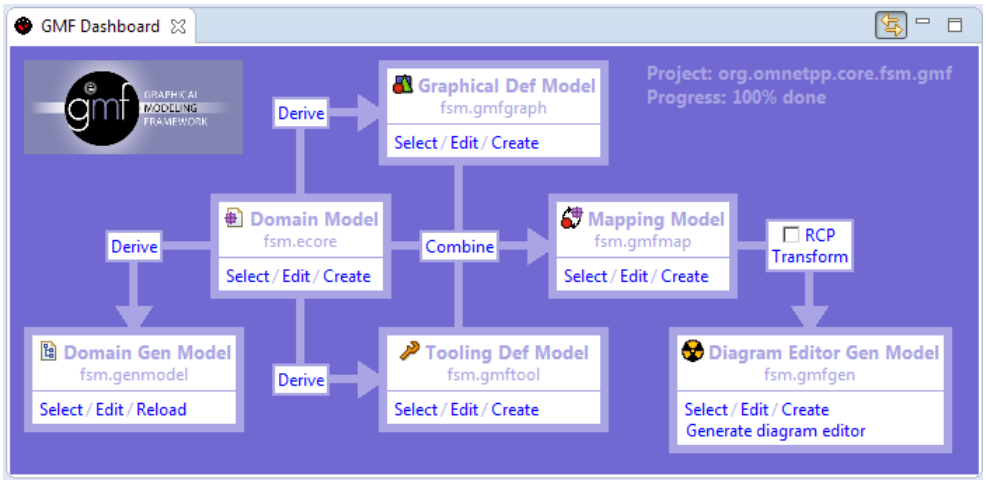

Abbildung 5.2.: GMF Dashboard View

5.1.1. EMF Domain Model (*.ecore / *.ecore_diagram)

Wie in Abschnitt 4.3.2 beschrieben wurde, gilt das *Domain Model* als Basis für die Erzeugung eines GMF Editors. Dieses wird in dem Eclipse Modeling Framework (EMF) als Ecoremodell bezeichnet. EMF bietet für die Bearbeitung eines Ecoremodells zwei Editoren an. Der erste Editor basiert auf der UML-Notation (**.ecore*) und der zweite ist ein Strukturbaum-basierter Editor (**.ecore_diagram*). Das *Domain Model* definiert die Struktur der OMNeT++ FSM und wurde mit Hilfe des zweiten Editors (**.ecore_diagram*) entwickelt (siehe Abbildung 5.3) [vgl. Engelmann, S. 7].

Wie in Abbildung 5.3 zu sehen ist, bietet der Ecore Editor eine Menge von Entitäten, die zur Modellierung von Ecoremodellen dienen (siehe Palette). Die genutzten Entitäten, die zum Aufbau des Ecoremodells der OMNeT++ FSM verwendet wurden, sind:

- ***EClass***: repräsentiert eine Klasse. Diese hat einen Namen sowie Referenzen und Attribute (z.B. die Klassen *FSM*, *InitialState*, *State* usw. in Abbildung 5.3).
- ***EAttribute***: repräsentiert ein Attribut einer Klasse *EClass* und besitzt einen Namen und einen Typ (z.B. die Attribute *Guard* und *Effect* der Klasse *Transition* in Abbildung 5.3).
- ***EDatatype***: bestimmt den Typ eines Attributs *EAttribute*.
- ***EReference***: repräsentiert Beziehungen zwischen den Klassen (Assoziation, Aggregation usw.) (siehe Abbildung 5.3).

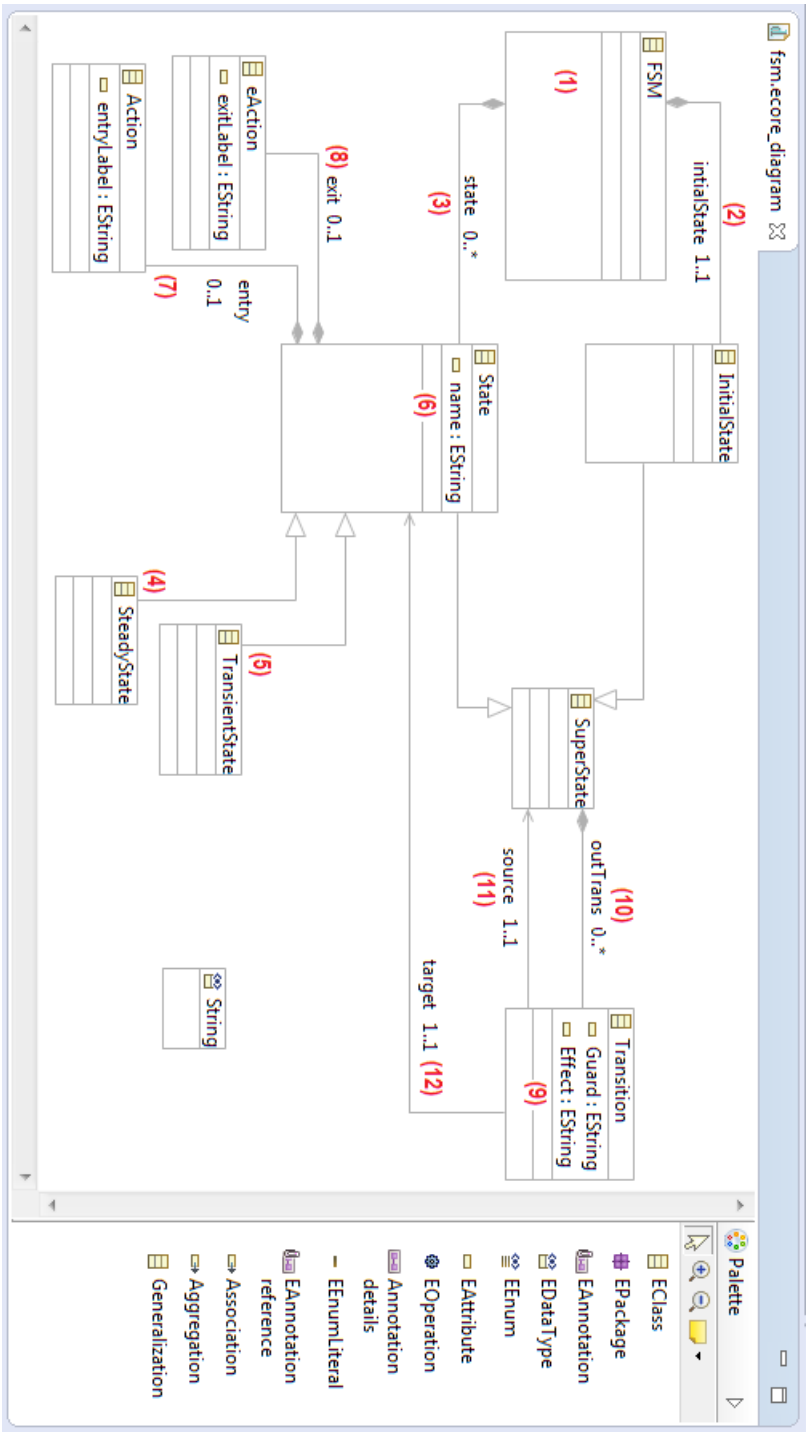

Abbildung 5.3.: Das mit Hilfe des EMF-Editors modellierte Ecoremodell der OMNeT++ FSM

Abbildung 5.3 zeigt die Elemente des hier entwickelten Ecoremodells:

- Die OMNeT++ FSM entspricht einer Klasse (*EClass:FSM*) (siehe (1)).
- Die OMNeT++ FSM (*EClass:FSM*) besitzt genau einen Initial-Zustand (*EClass:InitialState*) (siehe (2)).
- Die OMNeT++ FSM (*EClass:FSM*) kann beliebig viele normale Zustände (*EClass:State*) besitzen (siehe (3)).
- Ein normaler Zustand entspricht entweder einem Steady-Zustand (*EClass:SteadyState*) (siehe (4)) oder einem Transient-Zustand (*EClass:TransientState*) (siehe (5))).
- Ein Steady-/Transient-Zustand besitzt einen Namen (*EAttribute:name*) (siehe (6)).
- Steady-/Transient-Zustände können max. ein Entry-Label (*EClass:Action*) (siehe (7)) und max. ein Exit-Label (*EClass:eAction*) haben (siehe (8)).
- Die OMNeT++ FSM besitzt Transitionen (*EClass:Transition*). Jede Transition besitzt zwei Attribute, Guard (*EAttribute:Gurad*) und Effect (*EAttribute:Guard*) (siehe (9)).
- Initial-, Steady- und Transient-Zustände (EClass:SuperState) können beliebig viele ausgehende Transitionen besitzen (siehe (10)).
- Eine Transition besitzt genau einen Quellzustand. Ein Quellzustand kann ein Initial-, Steady- oder Transient-Zustand sein (siehe (11)).
- Ein Transition besitzt genau einen Zielzustand. Ein Zielzustand kann ein Steady- oder Transient-Zustand sein (d. h., ein Initial-Zustand besitzt keine eingehenden Transitionen) (siehe (12))

5.1.2. EMF Domain Generator Model (*.genmodel)

Nachdem im vorigen Abschnitt 5.1.1 beschrieben wurde, wie das Ecoremodell der OMNeT++ FSM mit Hilfe des EMF-Editors erstellt wurde, wird in diesem Abschnitt das EMF *Domain Generator Model* (*.genmodel) kurz erläutert.

Wie in Abschnitt 4.3.2 schon erwähnt, wird das *Domain Generator Model* aus dem *Domain Model* (*.ecore) abgeleitet (siehe Abbildung 4.4 unter dem Abschnitt 4.3 Seite 30). Dieser erzeugt aus dem *Domain Model* (*.ecore) Java-Klassen, die das Ecoremodell der OMNeT++ FSM beschreiben [vgl. Stahl, Thomas u. a., 2007, S.72]. Diese befinden sich im GMF Projekt *org.omnetpp.core.fsm.gmf* unter dem Ordner *src* (siehe Abbildung 5.4). Zusätzlich werden zwei Plugins erzeugt, *org.omnetpp.core.fsm.gmf.editor* und *org.omnetpp.core.fsm.gmf.editor* (siehe Abbildung 5.4). Diese stellen Wizards bereit, um Modellinstanzen zu erzeugen, und einen Texteditor, in dem die Modellinformationen eingetragen werden [vgl. Vogel, 2015, Abschnitt 5].

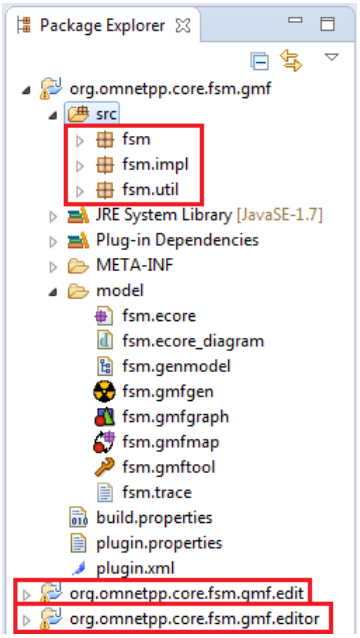

Abbildung 5.4.: EMF-Genratormodell erzeugter Code

5.1.3. GMF Graphical Definition Model (*.gmfgraph)

Das *Graphical Definition Model* (*.gmfgraph) wird ebenso aus dem *Domain Model* abgeleitet (siehe Abbildung 4.4 unter dem Abschnitt 4.3 Seite 30). In diesem werden die graphischen Elemente definiert, die für die Visualisierung des OMNeT++ FSM Modells verwendet werden [vgl. Brambilla u. a., 2012, S.97]. Das *Graphical Definition Model* lässt sich in Form einer Baumstruktur darstellen. Die Wurzel des Baumes bildet ein Canvas-Objekt. Innerhalb des Canvas werden die graphischen Elemente als Kinderknoten definiert. Beispielsweise wird in dem *Graphical Definition Model* festgelegt, dass ein Steady-Zustand die Form eines abgerundetes Rechteck hat und die Vordergrundfarbe Weiß hat usw. (siehe Abbildung 5.5).

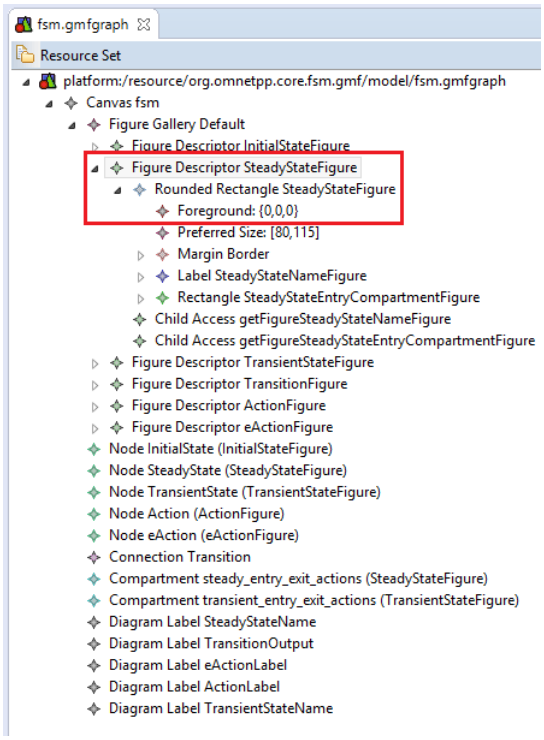

Abbildung 5.5.: *Graphical Definition Model* der OMNeT++ FSM

5.1.4. GMF Tooling Definition Model (*.gmftool)

Das *Tooling Definition Model* (*.gmftool) wird ebenfalls aus dem *Domain Model* abgeleitet (siehe Abbildung 4.4 unter dem Abschnitt 4.3 Seite 30). Wie schon erwähnt werden in diesem die Werkzeuge festgelegt, mit denen die FSM erzeugt bzw. bearbeitet wird. Diese werden über eine Tool-Palette bereitgestellt. Abbildung 5.6 zeigt, dass sich das *Tooling Definition Model* ebenso in der Form einer Baumstruktur darstellen lässt. In der Tool-Palette sind die Werkzeuge in drei Kategorien unterteilt:

- ***States***: In dieser Kategorie befinden sich die Werkzeuge, die den Zuständen der FSM entsprechen: *Initial*, *Steady* und *Transient*.
- ***Actions***: In dieser Kategorie befinden sich die *Entry-* und *Exit-Labels*.
- ***Connections***: In dieser Kategorie befindet sich das Werkzeug *Transition* (Siehe erzeugte Palette in Abbildung 5.8 Seite 44).

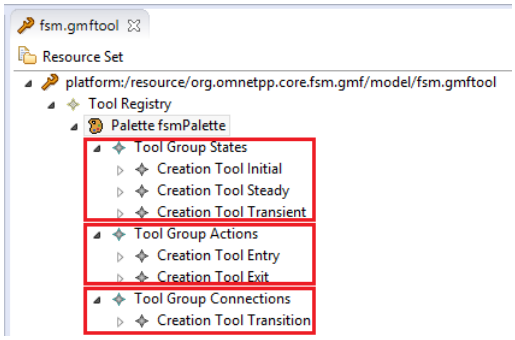

Abbildung 5.6.: GMF *Tooling Definition Model* der OMNeT++ FSM

5.1.5. GMF Mapping Definition Model (*.gmfmap)

Das *Mapping Definition Model* führt die vorigen Modelle (*.ecore*, *.gmfgraph* und *.gmftool*) zusammen (siehe Abbildung 4.4 unter dem Abschnitt 4.3 Seite 30). Es ordnet Elementen aus dem Ecoremodell ein graphisches Element (aus *.gmfgraph*) und ein Werkzeug (aus *.gmftool*) zu, um ein geeignetes *Mapping Definition Model* zu erzeugen.

5.1.6. GMF Diagram Editor Generation Model (*.*gmfgen*)

Wie schon erwähnt wird das *Diagram Editor Definiton Model* durch eine Transformation mit Hilfe des *GMF Dashboards* erzeugt. Dieses Modell dient als Konfigurationsdatei für die Generierung des graphischen Editors. Die Generierung wird durchgeführt, indem im *GMF Dashboard* auf *Generate diagram editor* geklickt wird (siehe Seite 36 Abbildung 5.2 rechts unten). Dementsprechen wird ein weiteres Plugin *org.omnetpp.core.fsm.gmf.diagram* erzeugt (siehe Abbildung 5.7), das zusammen mit den EMF generierten Plugins den OFGE bildet.

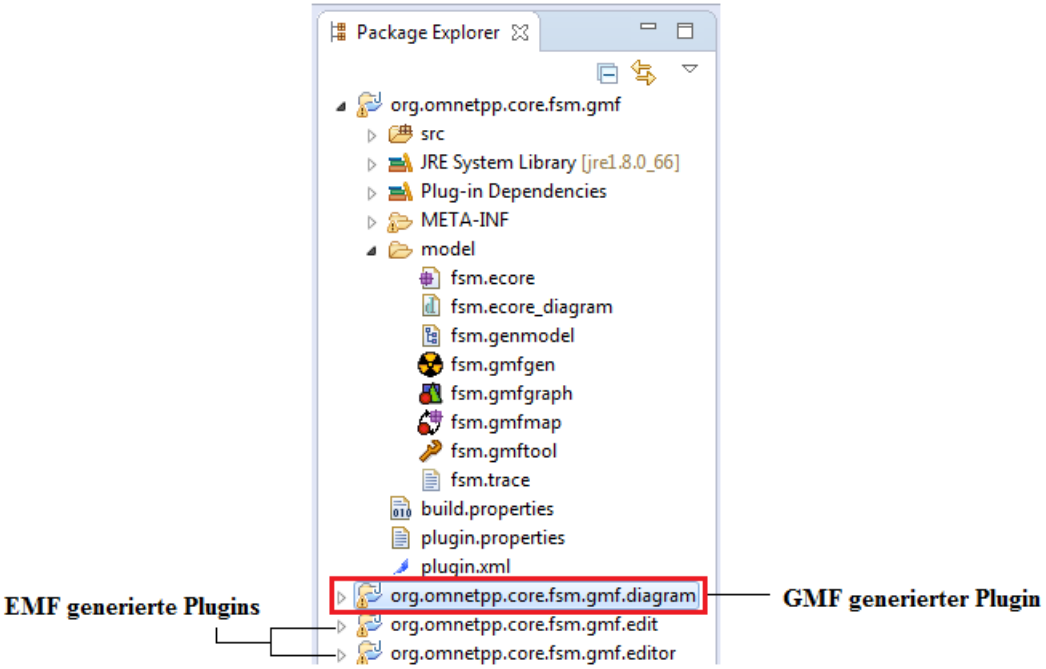

Abbildung 5.7.: GMF *Diagram Generation Model* erzeugter Code

In Abbildung 5.8 wurde als Beispiel der Synchronisation Client (SC) aus dem Abschnitt 2.4.1 Seite 13 mit Hilfe des OMNeT++ FSM graphischen Editors modelliert. Wie in Abbildung 5.8 zu sehen, entspricht der Editor einer *.fsm-Datei, die eine graphische Umgebung für die Gestaltung von OMNeT++ FSMs bietet.

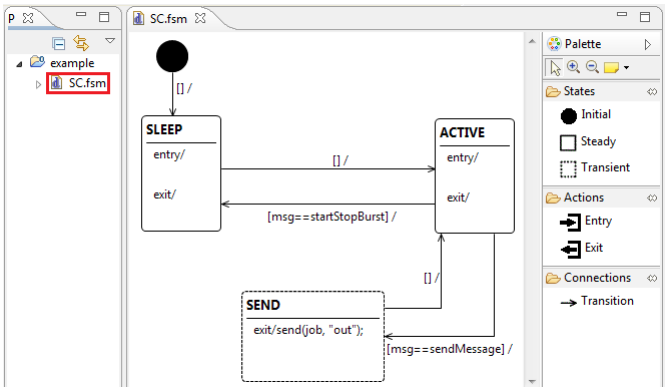

Abbildung 5.8.: Die SC FSM mit Hilfe des OFGE modelliert.

5.1.7. XML-Datei des OFGE

Wie in Abschnitt 4.1 beschrieben wurde, bietet GMF für jede FSM, die mittels OFGE modelliert wurde, eine XML-Datei an. Diese enthält die Informationen zur graphischen Darstellung der modellierten FSM (z.B. welche Zustände vorhanden sind und jeweils die Position, welche Transitionen es gibt usw.). Die XML-Datei wird benötigt, um die FSM im OFGE einlesen zu können. Zusätzlich enthält diese Informationen, die für die Codegenerierung im nächsten Abschnitt 5.2 nötig sind. Aus diesem Grund wird in diesem Abschnitt der Aufbau der XML-Datei dargestellt.

Aufbau der XML-Datei

In Listing 5.1 wird die XML-Datei der modellierten FSM in Abbildung 5.8 dargestellt. Wie zu sehen ist, beginnt die XML-Datei des OFGE mit einem Wurzelknoten ***xmi:XMI*** (Zeile 2). Dieser enthält zwei Unterbäume, ***fsm:FSM*** (Zeile 5) und ***notation:Diagram*** (Zeile 32). Der erste Unterbaum ***fsm:FSM*** enthält die Instanzen der modellierten FSM (Zustände, Transitionen usw.). Der zweite Unterbaum ***notation:Diagram*** enthält die Darstellungsinformationen der Instanzen (Position, Größe usw.). Nicht alle Daten der XML-Datei sind für die Codegenerierung relevant. Daher werden die relevanten Daten der Unterbäume (***fsm:FSM*** und ***notation:Diagram***) jeweils als Baumstruktur im weiteren Verlauf verständlicher dargestellt.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI ... >

  <!-- Der erste Unterbaum fsm:FSM -->
  <fsm:FSM xmi:id="_zfQ">

    <!-- SLEEP Zustand vom Typ Steady -->
    <state xmi:type="fsm:SteadyState" xmi:id="_2Av" name="SLEEP">
      <outTrans xmi:type="fsm:Transition" xmi:id="_6qE" target="_3Cb" source="_2Av"/>
      <entry xmi:type="fsm:Action" xmi:id="_Kyd"/>
      <exit xmi:type="fsm:eAction" xmi:id="_LJ9"/>
    </state>

    <!-- Weiter state Knoten -->
    ...

    <!-- SEND Zustand vom Typ Transient -->
    <state xmi:type="fsm:TransientState" xmi:id="_34i" name="SEND">
      <outTrans xmi:type="fsm:Transition" xmi:id="_9kt" target="_3Cb" source="_34i"/>
      <exit xmi:type="fsm:eAction" xmi:id="_PJy" exitLabel="send(job,___&amp;quot;out&amp;quot;);"/>
    </state>

    <!-- Initial Zustand -->
    <intialState xmi:type="fsm:InitialState" xmi:id="_1rf">
      <outTrans xmi:type="fsm:Transition" xmi:id="_5hY" target="_2Av" source="_1rf"/>
    </intialState>

  </fsm:FSM>


  <!-- Der Zweite Unterbaum notation:Diagram  -->
  <notation:Diagram ... >

   <!-- children knoten, der die Darstellungsinformationen eines Zustandes enthält (Position, Größe usw.) -->
   <children xmi:type="notation:Node" xmi:id="_1ru" type="2007" element="_1rf">
    <styles xmi:type="notation:DescriptionStyle" xmi:id="_1ru"/>
    <styles xmi:type="notation:FontStyle" xmi:id="_1ru" fontName="Segoe_UI"/>
    <layoutConstraint xmi:type="notation:Bounds" xmi:id="_1ru" x="27" y="8" width="30" height="30" />
   </children>
   <!-- Weiter children Knoten -->
   ...

   <!-- edges knoten, der die Darstellungsinformationen einer Transition enthält -->
   <edges xmi:type="notation:Edge" xmi:id="_5he" type="4001" element="_5hY" source="_1ru" target="_2Ax">
     <children xmi:type="notation:DecorationNode" xmi:id="_5hg" type="6001">
      <layoutConstraint xmi:type="notation:Location" xmi:id="_5hg" x="-7" y="-12"/>
     </children>
     <styles xmi:type="notation:RoutingStyle" xmi:id="_5he"/>
     <styles xmi:type="notation:FontStyle" xmi:id="_5he" fontName="Segoe_UI"/>
     <bendpoints xmi:type="notation:RelativeBendpoints" xmi:id="_5he" points="[9,22,-23,-55]$[23,57,-9,-20]"/>
     <targetAnchor xmi:type="notation:IdentityAnchor" xmi:id="_5h5" id="(0.38,0.008)"/>
     <sourceAnchor xmi:type="notation:IdentityAnchor" xmi:id="_8Qn" id="(0.76,0.19)"/>
   </edges>
   <!-- Weiter edges Knoten -->
   ...

  </notation:Diagram>

</xmi:XMI>
```

Listing 5.1: Beispiel für die XML-Datei einer OFGE FSM (*SC.fsm*) (siehe Abbildung 5.8)

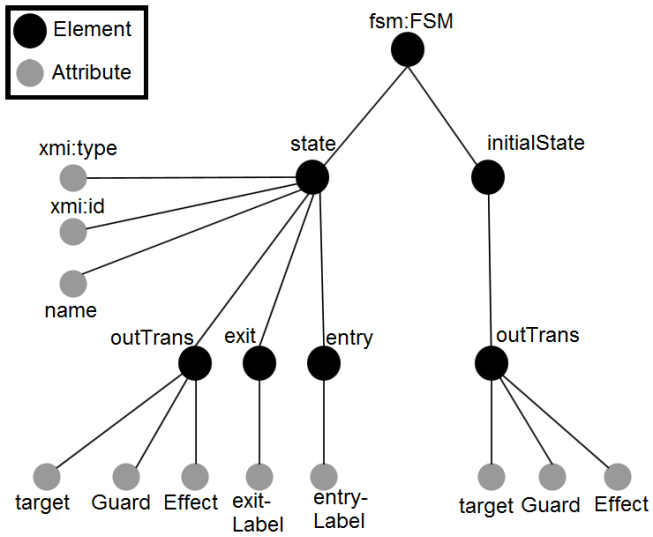

Abbildung 5.9.: Die Baumstruktur des Unterbaums ***fsm:FSM*** aus der XML-Datei des OFGE

Wie in Abbildung 5.9 zu sehen ist, besitzt der Unterbaum ***fsm:FSM*** die Unterknoten ***state*** und ***initialState***. Diese haben eine Menge von untergeordneten Knoten (Attribute bzw. Elemente), die für die Codegenerierung relevant sind:

- Attribut ***xmi:type***: Um welchen Zustandstyp es sich handelt (Steady, Transient).
- Attribut ***xmi:id***: Eine eindeutige generierte ID für jeden Zustand.
- Attribut ***name***: Zustandsname.
- Element ***outTrans***: Dies entspricht einer ausgehenden Transition und besitzt wiederum drei relevante Attribute:
  - Attribut ***target***: ID des Zielzustandes der Transition.
  - Attribut ***Effect***: Übergangseffekt.
  - Attribut ***Guard***: Eine Transition kann nur durchlaufen werden, wenn der Wächterausdruck (*Guard*) wahr ist.
- Element ***entry***/***exit***: Diese Element besitzt ein relevantes Attribut:

- Attribut ***entryLabel/exitLabel***: Aktionen, die beim Eintreten/Verlassen eines Zustandes stattfinden.

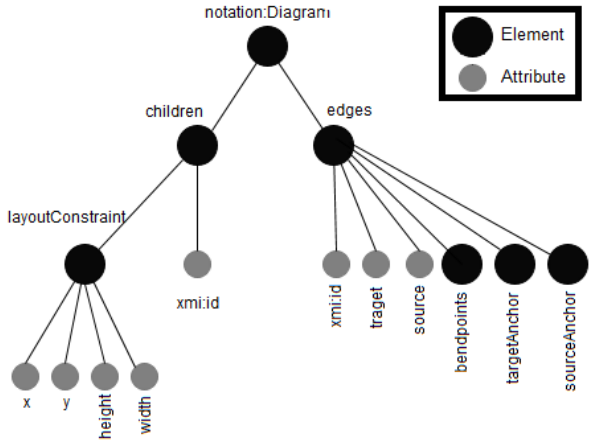

Abbildung 5.10.: Die Baumstruktur des Unterbaums ***notation:Diagram*** aus der XML-Datei des OFGE

Wie in Abbildung 5.10 zu sehen ist, besitzt der Unterbaum ***notation:Diagram*** die Unterknoten ***children*** und ***edges***. Diese haben eine Menge von untergeordneten Knoten (Attribute bzw. Elemente), die für die Codegenerierung relevant sind:

- Element ***children***: Für jeden Zustand in der modellierten FSM gibt es einen Knoten ***children***, der die Darstellungsinformationen des Zustandes beschreibt. Folgende relevante Attribute besitzt der Knoten ***children***:
  - Attribut ***xmi:id***: ID des Zustandes.
  - Attribute ***x*** und ***y***: Position des Zustandes.
  - Attribute ***height*** und ***width***: Größe des Zustandes.
- Element ***edges***: Für jede Transition in der modellierten FSM gibt es einen Knoten ***edges***, der die Darstellungsinformationen der Transition beschreibt. Folgende relevante untergeordnete Knoten besitzt der Knoten ***edges***:
  - Attribut ***xmi:id***: ID der Transition.

- Attribut ***source*/*target***: ID des Quell-/Zielzustandes.
- Element ***bendpoints***: Punkte, die durch eine Transition miteinander verbunden sind.
- Element ***targetAnchor/sourceAnchor***: Die Stelle, an der sich eine Transition an dem Quell-/Zielzustandes verankert.

5.2. XSLT

In diesem Abschnitt wird erläutert, wie die Codegenerierung mit Hilfe der Extensible Stylesheet Language Transformation (XSLT) [World Wide Web Consortium, 2007] abgewickelt wurde. Als Beispiel wird weiterhin der Synchronisation Client (SC) aus den vorigen Abschnitten verwendet (siehe Abbildung 5.8).

Wie in Abschnitt 4.4 schon beschrieben, wird für jede XSL-Transformation eine Quelldatei und ein XSLT-Stylesheet benötigt, um die gewünschte Zieldatei zu erzeugen. Im Rahmen dieser Arbeit werden drei Transformationen mit derselben Quelldatei durchgeführt. Die Quelldatei entspricht der XML-Datei einer OFGE FSM (siehe Abschnitt 5.1.7). Die XML-Datei wird durch drei XSLT-Stylesheets in die drei Zieldateien transformiert

1. Eine ***.cc**-Datei:
   - entspricht dem OMNeT++ Modul, für das die FSM mittels OFGE modelliert wurde.
   - beschreibt die OMNeT++ FSM in der C++ Programmiersprache.
2. Eine ***_fsm_updater.cc**-Datei:
   - entspricht ebenso einem OMNeT++ Modul.
   - hat die Aufgabe, die modellierte FSM zur Laufzeit in der OMNeT++ Simulation darzustellen und zu aktualisieren.
3. Eine ***.ned**-Datei:
   - beschreibt den Aufbau der oben genannten Module in der NED-Sprache.

Die drei aufgezählten erzeugten Zieldateien werden in den Abschnitten 5.2.2, 5.2.3 und 5.2.4 ausführlicher vorgestellt.

5.2.1. Beispiel einer XSL-Transformation

In diesem Abschnitt wird eine kleine Transformation durchgeführt, um zu zeigen, wie anhand von Umwandlungsregeln eines XSLT-Stylesheets aus XML-Code beliebiger Text erzeugt wird. Als Beispiel wird ein Teil der ersten Transformation durchgeführt, die den C++ Code für die modellierte OMNeT++ FSM erzeugt. Das Beispiel entspricht der Erzeugung eines Enums, das die Zustände der modellierten FSM definiert (siehe Abschnitt 2.4.1 Seite 12).

Wie in Abschnitt 4.4 (Seite 32) beschrieben wurde, enthält ein XSLT-Stylesheet verschiedene Elemente. Das Element *<xsl:template>* ist eines der wichtigsten Elemente. Bongers beschreibt dies wie folgt: *„Die Deklaration xsl:template dient zur Definition einer Templateregel bzw. eines benannten Templates. Mit Hilfe des in diesem Template enthaltenen Sequenzkonstrukutors kann in Zusammenhang mit einem verarbeiteten Node ein beliebiger Output in einen Ergebnisbaum erzeugt werden. “* [Bongers, 2004, S. 863]. In Listing 5.2 wird ein Codeausschnitt aus dem XSLT-Stylesheet der ersten Transformation dargestellt, das für die Erzeugung des Enums zuständig ist (vollständiges XSLT-Stylesheet ist unter *org.omnetpp.core.fsm.gmf.diagram/src/fsm/diagram/part/transform1.xsl* zu finden). In diesem wird der erste Unterbaum der XML-Datei (siehe XML-Unterbaum in Listing 5.1 und Unerbaumstruktur in Abbildung 5.9) ***fsm:FSM*** mit Hilfe von *<xsl:template>* verarbeitet:

- Text im Template wird vom XSLT-Prozessor in die Zieldatei übernommen. Somit wird z.B. Zeile 2-5 in die Zieldatei geschrieben (siehe Listing 5.3 Zeile 1-4).
- Mit einer *for-each*-Schleife werden innerhalb des Unterbaums ***fsm:FSM*** für jeden Zustand vom Typ Steady der Name und die Knotenposition (die als ID genutzt wird) in die Zieldatei geschrieben (siehe Listing 5.3 Zeile 5-6).
- Mit einer *for-each*-Schleife werden innerhalb des Unterbaums ***fsm:FSM*** für jeden Zustand vom Typ Transient ebenso der Name und die Knotenposition in die Zieldatei geschrieben (siehe Listing 5.3 Zeile 7).

```
<xsl:template match="fsm:FSM">
    cFSM fsm;     //OMNeT++ FSM instance
    enum {        //Defining states of the FSM
          INIT = 0,
          //Zustandsname = FSM_Typ(ID)
    <xsl:for-each select="state[@xmi:type='fsm:SteadyState']">
       <xsl:value-of select="@name"/> = FSM_Steady(<xsl:value-of select="position()"/>),
    </xsl:for-each>

    <xsl:for-each select="state[@xmi:type='fsm:TransientState']">
       <xsl:value-of select="@name"/> = FSM_Transient(<xsl:value-of select="position()"/>),
    </xsl:for-each>
    };
 </xsl:template>
```

Listing 5.2: Codeausschnitt aus dem XSLT-Stylesheet der ersten Transformation

```
cFSM fsm;     //OMNeT++ FSM instance
enum {        //defining states of the FSM
       INIT = 0,
       //state name = FSM_type(ID)
       ModifyingMsg = FSM_Steady(1),
       WaitingForMsg = FSM_Steady(2),
       ReceivingMsg = FSM_Transient(1),
};
```

Listing 5.3: Transformationsausgabe des Codeausschnitts aus Listing 5.2

5.2.2. Erste XSL-Transformation (*.cc)

In diesem Abschnitt wird die Zieldatei der ersten XSL-Transformation vorgestellt. Wie schon erwähnt, wird als Quelldatei für diese Transformation die XML-Datei einer OFGE FSM verwendet. Als Beispiel wird hier die XML-Datei des Synchronisation Client (SC) verwendet (siehe Listing 5.1 Seite 45). Das XSLT-Stylesheet, das für diese Transformation geschrieben worden ist, erzeugt mit Hilfe der XML-Quelldatei die *SC.cc*-Zieldatei (siehe Stylesheet unter *org.omnetpp.core.fsm.gmf.diagram/src/fsm/ diagram/part/transform1.xsl*).

```
1  class SC: public cSimpleModule{
2      cFSM fsm;     //OMNeT++ FSM instance
3      enum {        //defining states of the FSM
4         INIT = 0,
5         //state name = FSM_type(ID)
6         SLEEP = FSM_Steady(1),
7         ACTIVE = FSM_Steady(2),
8         SEND = FSM_Transient(1),
9      };
10     //variables used to update the displayed FSM during runtime
11     cMessage* updateMsg;
12     String nameOfUsedTransition;
13
14 public:
15     SC();
16     virtual ~ SC();
17
18 protected:
19     virtual void initialize();
20     virtual void handleMessage(cMessage *msg);
21     virtual void sendUpdateMsg(char* nameOfCurrentState);
22     virtual void sendTransitionUpdateMsg();
23 };
```

Listing 5.4: Codeausschnitt aus der erzeugten *SC.cc*-Zieldatei der ersten XSL-Transforamtion

Listing 5.4 zeigt einen Codeausschnitt aus der erzeugten *SC.cc*-Zieldatei. Wie zu sehen ist, enthält diese die C++ Implementierung des SC-Moduls, für das die FSM modelliert wurde (siehe Abbildung 5.8). Das Modul enthält ein Enum, das die Zustände der FSM definiert (siehe Zeile 3-9). Zusätzlich besitzt dies vier Methoden (siehe Zeile 19-22). In der *handlemessage()* Methode befindet sich die OMNeT++ FSM des SC-Moduls (siehe Abschnitt 2.4.1). Die Methoden *sendUpdateMsg()* und *sendTransitionUpdateMsg()* werden von der *handlemessage()* Methode bei jedem Zustandswechsel der FSM aufgerufen.

Diese senden eine Nachricht an das Modul, das für die Aktualisierung der zur Laufzeit dargestellten FSM verantwortlich ist (siehe Abschnitt 5.2.3), um dementsprechend die Darstellung der FSM zu aktualisieren.

5.2.3. Zweite XSL-Transformation (*_fsm_updater.cc)

In diesem Abschnitt wird die Zieldatei der zweiten XSL-Transformation vorgestellt. Wie schon erwähnt, wird als Quelldatei für die zweite XSL-Transformation wird ebenso die XML-Datei einer OFGE FSM verwendet. Als Beispiel wird hier die XML-Datei des Synchronisation Client (SC) verwendet (siehe Listing 5.1 Seite 45). Das XSLT-Stylesheet, das für diese Transformation geschrieben wurde, erzeugt mit Hilfe der XML-Quelldatei eine *SC_fsm_updater.cc* Zieldatei (siehe Stylesheet unter *org.omnetpp.core.fsm.gmf.diagram/-src/fsm/diag ram/part/transform2.xsl*).

```
1  class SC_fsm_updater: public cSimpleModule {
2     private:
3          //states
4          cRectangleFigure *init;
5          cRectangleFigure *SLEEP;
6          cRectangleFigure *ACTIVE;
7          cRectangleFigure *SEND;
8          //transitions and their labels
9          cPolylineFigure *t1;
10         cLabelFigure *t1Label;
11         cPolylineFigure *t2;
12         cLabelFigure *t2Label;
13         cPolylineFigure *t3;
14         cLabelFigure *t3Label;
15         cPolylineFigure *t4;
16         cLabelFigure *t4Label;
17         cPolylineFigure *t5;
18         cLabelFigure *t5Label;
19
20         Vector bendpointsVector;      //holds calculated bendpoints of a spesific transition
21         //holds required information to calculate the real bendpoints of each transition
22         struct xmlData { .. }
23
24         //variables used
25         String nameOfCurrentState;
26         cRectangleFigure *lastState;
27         cPolylineFigure *lastTransition;
28         bool firstTransitionUpdateCall = true;
29         bool firstStateUpdateCall = true;
30
31    protected:
32         virtual void initialize();
33         virtual void handleMessage(cMessage *msg);
34         void updateStateLayout(cRectangleFigure* figure);
35         void updateTransitionLayout(cPolylineFigure* figure);
36         Vector getRealBendpoints(xmlData par);
37 };
```

Listing 5.5: Codeausschnitt aus der erzeugten *SC_fsm_updater.cc* Zieldatei der zweiten XSL-Transforamtion

Listing 5.5 zeigt einen Codeausschnitt aus der erzeugten *SC_fsm_updater.cc* Zieldatei. Wie zu sehen ist, enthält diese die C++ Implementierung des Moduls, das für die Darstellung und Aktualisierung der FSM zur Laufzeit zuständig ist. Wie in Abschnitt 4.5 beschrieben wurde, wird die Darstellung der FSM mit Hilfe der Canvas API abgewickelt. Jede FSM-Instanz (Zustand, Transition usw.) entspricht einer Figur, die von der Canvas API bereitgestellt wird. Die Figuren werden in der **.ned*-Datei erzeugt (siehe Abschnitt 5.2.4). Diese können von der **.cc*-Datei angesprochen werden, indem für jede Figur ein Objekt vom Typ **Figure* erzeugt wird (siehe Zeile 4-18). Somit können die Figuren über die Objektinstanzen zur Laufzeit angesprochen und dementsprechend manipuliert werden (Farbe, Position ändern usw.).

Außerdem besitzt das Modul fünf Methoden (siehe Zeile 32-36). In der *initialize()* Methode werden die Figuren initialisiert. In der *handleMessage()* Methode werden Nachrichten vom Modul der ersten Transformation empfangen, die für die Aktualisierung der FSM zuständig sind (siehe Abschnitt 5.2.2). Diese werden ausgewertet und dementsprechend wird *updateStateLayout()* bzw. *updateTransitionLayout()* aufgerufen. Somit wird das Layout eines Zustandes bzw. einer Transition geändert.

Um die Transitionen zu zeichnen, werden so genennte Bendpoints[1] benötigt. Die Werte der Bendpoints, die in der XML-Quelldatei stehen, sind relativ. Die *getRealBendpoints()* Methode berechnet aus den relativen Werten die absoluten. Diese wird von der *initialize()* Methode aufgerufen, um bei der Initialisierung die Transitionen zu zeichnen.

5.2.4. Dritte XSL-Transformation **.ned*

In diesem Abschnitt wird die Zieldatei der dritten XSL-Transformation vorgestellt. Wie schon erwähnt, wird als Quelldatei für die dritte XSL-Transformation ebenso die XML-Datei des Synchronisation Client (SC) verwendet (siehe Listing 5.1 Seite 45). Das XSLT-Stylesheet, das für diese Transformation geschrieben wurde, erzeugt mit Hilfe der XML-Quelldatei die *SC.ned*-Datei (siehe Stylesheet unter *org.omnetpp.core.fsm-.gmf.diagram/src/fsm/diagram/part/transform3.xsl*).

[1] Punkte, die durch eine Transition miteinander verbunden sind.

```
1  //the module that displays the FSM during runtime
2  simple SC_fsm_updater
3  {
4      @display("bgb=1,1,black;i=block/fork;p=42,147;");
5
6      gates:
7          input update_signal @directIn;
8  }
9
10 //the main module which contains the OMNeT++ FSM
11 simple SC
12 {
13     @display("i=block/routing;p=42,65;");
14
15     gates:
16             input in;
17             output out;
18 }
19
20 //compound module which holds the above-defined modules and all the canvas parameters
21 module SC_compound
22 {
23     parameters:
24
25     @display("bgb=1100,615,white;i=block/routing;");
26
27     //intital state and its outgoing transitions
28     @figure[init](type=group; transform=translate());
29     @figure[init.layout](type=rectangle; pos=0,0; size=30,30;
30     cornerRadius=60; lineColor=#000000; fillColor=black);
31     @figure[init.outTrans1](type=polyline; endArrowhead=barbed);
32     @figure[init.outTrans1Label](type=label; text="[]/"; font=Arial,12;
33      color=black; anchor=center);
34
35     //figures of the state "SLEEP" and its outgoing transitions
36     @figure[SLEEP](type=group; transform=translate(295,225));
37     @figure[SLEEP.layout](type=rectangle; pos=0,0; size=80,115;
38      cornerRadius=7; lineColor=#000000; fillColor=white);
39
40     .
41     .
42     .
43
44     gates:
45         input in;
46         output out;
47     submodules:
48         fsm: SC_fsm_updater;
49         SC: SC;
50     connections:
51         in --> SC.in;
52         SC.out --> out;
53 }
54
55 network net
56 {
57     submodules:
58         compound_module: SC_compound;
59 }
```

Listing 5.6: Codeausschnitt aus der erzeugten Zieldatei (*SC.ned*) der dritten XSL-Transforamtion

Die *SC.ned*-Zieldatei beschreibt in der NED-Sprache den Aufbau der Module, die in der ersten und zweiten XSL-Transformation erzeugt wurden (siehe Abschnitt 5.2.2 und 5.2.3). Wie in Listing 5.6 zu sehen ist, entspricht das *SC_fsm_updater*-Modul einem Simple Modul, das ein *directIn*-Gate besitzt. Über dieses werden die Aktualisierungsnachrichten vom *SC*-Modul direkt empfangen (siehe Zeile 2-8). Das *SC*-Modul entspricht ebenso einem Simple Modul, das standardmäßig zwei Gates (*in* und *out*) besitzt (siehe Zeile 11-18).

Die oben genannten Module *SC* und *SC_fsm_updater* werden als Submodule zu dem Compound Modul *SC_compound* hinzugefügt (siehe Zeile 47-49). Dies enthält die Figuren, die für die Darstellung der FSM zur Laufzeit nötig sind. Diese werden über die *@figure* Property[1] (z.B. Zeile 28) erzeugt (siehe Abschnitt 4.5). Das Compound Modul *SC_compound* besitzt zwei Gates (*in* und *out*), die mit den Gates des *SC*-Moduls verbunden sind (siehe Zeile 50-52). Somit können Module (z.B. *SC*), für die eine FSM mittels OFGE modelliert wurde, über deren Compound Modul (z.B. *SC_compound*) angesprochen werden.

[1] Im Allgemeinen werden @-Wörter, wie z.B. @figure, in NED Properties genannt. Diese werden verwendet, um verschiedene Objekte mit Metadaten zu annotieren. Properties können Dateien, Module, Parameter, Gates, Connections usw. zugewiesen werden [vgl. Manual OMNeT++ Community, b, Abschnitt 3.12].

5.3. Zusammenführung von OFGE und XSLT

Ein Ziel dieser Arbeit ist es, den Implementierungscode einer FSM automatisch mittels XSLT zu generieren, nachdem der Nutzer mittels OFGE diese FSM modelliert hat. Um dies zu erreichen, muss die Implementierung des OFGE (siehe Abschnitt 5.1) und der XSLT (siehe Abschnitt 5.2) zusammengeführt werden. In diesem Abschnitt wird die Zusammenführung der beiden Ansätze erläutert.

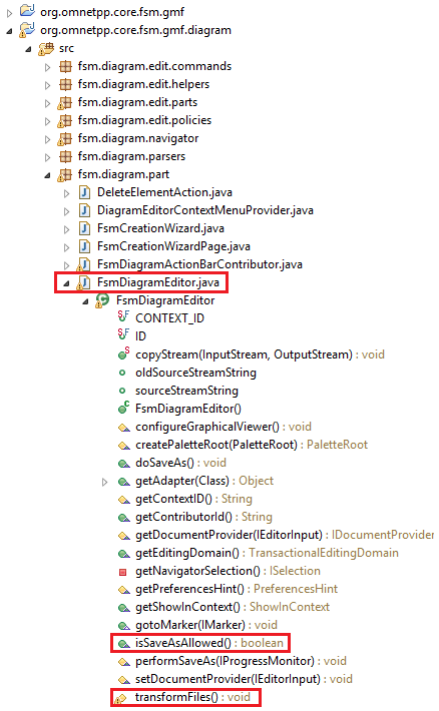

Abbildung 5.11.: Übersicht über das Projekt *org.omnetpp.core.fsm.gmf.diagram* und dessen *FsmDiagramEditor.java* Datei

Wie in Abschnitt 5.1 beschrieben wurde, wird beim Entwickeln eines graphischen Editors mittels GMF eine Menge von Projekten erzeugt, die zusammen den OFGE bilden (siehe vier Projekte *org.omnetpp.core.fsm.gmf.** in Abbildung 5.7 Seite 43). In einem dieser Projekte *org.omnetpp.core.fsm.gmf.diagram* befindet sich eine Datei unter dem Pfad *src/fsm/diagram/part/FsmDiagramEditor.java* (siehe Abbildung 5.11). Wie zu sehen

ist, enthält diese eine Methode namens *isSaveAsAllowed()*. Diese wird bei der Speicherung der *.fsm-Datei des OFGE, in der die gewünschte FSM modelliert wird (siehe Beispiel Abbildung 5.8), automatisch aufgerufen. Dementsprechend wird in der *FsmDiagramEditor.java* Datei die *transformFiles()* Methode hinzugefügt. Diese ist für die Codegenerierung mittels XSLT zuständig. Die *transformFiles()* wird von *isSaveAsAllowed()* aufgerufen, sodass bei jeder Speicherung der FSM die XSLT-Codegenerierung statt findet und somit die drei XSLT-Transformationen durchgeführt werden (siehe einzelne Transformationen unter den Abschnitten 5.2.2, 5.2.3 und 5.2.4).

Da die *transformFiles()* Methode umfangreich ist, wird in Listing 5.7 ein Ausschnitt aus dieser Methode dargestellt. Dieser umfasst die Grundfunktionalität der *transformFiles()* Methode. Die vollständige Implementierung dieser Methode befindet sich unter der Datei *org.omnetpp.core.fsm.gmf.diagram/src/fsm/diagram/part/FsmDiagramEdi- tor.java*.

```
protected void transformFiles() {
    //get path of the OFGE (OMNeT++ FSM Graphical Editor) (*.fsm file).
    IPathEditorInput inp = (IPathEditorInput) getEditorInput();
    String sourcePath = (inp.getPath()).toString();
    /*
     * Creating a stream which corresponds to the XML file of the OFGE. This stream will be
     * used as a source stream for the XSL transformations.
     */
    Source source = new StreamSource(new File(sourcePath.toString()));
    //save string of the path where the each of the generated files will be stored
    String sourcePathString = sourcePath.toString();
    String targetPath = sourcePathString.substring(0,sourcePathString.lastIndexOf("."));
    /*
     * Creating the stream that corresponds to the generated files (*.cc, *fsm_updater.cc and
     *.ned). These streams will be used as a target stream for the XSL transformations.
     */
    Result result1 = new StreamResult(new File(targetPath + ".cc"));
    Result result2 = new StreamResult(new File(targetPath + "_fsm_updater.cc"));
    Result result3 = new StreamResult(new File(targetPath + ".ned"));
    ...
    //if FSM has been changed --> regenerate the three files
    if ((!sourceStreamString.equals(oldSourceStreamString))&& (!(oldSourceStreamString == ""))) {
        ...
        //transformations declaration
        Transformer transformer1 = factory1.newTransformer(xslt1);
        Transformer transformer2 = factory2.newTransformer(xslt2);
        Transformer transformer3 = factory3.newTransformer(xslt3);
        ...
        //transform files
        transformer1.transform(source, result1);
        transformer2.transform(source, result2);
        transformer3.transform(source, result3);
        ...
    }
}
```

Listing 5.7: Codeausschnitt aus der hinzugefügten Methode *transformFiles()* aus der *FsmDiagramEditor.java* Datei

Wie in Abschnitt 4.4 beschrieben wurde, benötigt jede XSL-Transformation drei Dateien: das XSLT-Stylesheet, die Quell- und Zieldatei. Bei der Implementierung einer XSL-Transformation entspricht das XSLT-Stylesheet einem Objekt vom Typ *Transformer*. Um die XSL-Transformation durchzuführen, wird vom *Transformer* Objekt die folgende Methode aufgerufen:

***void** transform(**Source** srcFile, **Result** rsltFile);*

Diese Methode erhält als ersten Parameter einen Source Stream, der der Quelldatei entspricht, und als zweiten Parameter einen Result Stream, der der Zieldatei entspricht, in die der erzeugte Code geschrieben wird.

Als erstes wird der Ort (Pfad) festgelegt, in dem sich die OFGE FSM befindet (siehe Zeile 3-4). Dieser wird benötigt um:

- an die Quelldatei heranzukommen. Diese entspricht der XML-Datei des OFGE.
- den Pfad der erzeugten Zieldateien angeben zu können, da diese am selben Ort erzeugt werden, an dem sich die OFGE FSM befindet.

Als nächstes wird der Source Stream erzeugt, der der XML-Datei der OFGE FSM entspricht (siehe Zeile 9). Wie schon erwähnt, gilt dieser als Quelldatei für die drei XSL-Transformation.

In Zeile 17-19 werden die Result Streams der drei XSL Transformationen erzeugt. Diese entsprechen den Zieldateien, indem der erzeugte Code reingeschrieben wird.

Darauffolgend wird für jede der drei XSL-Transformationen ein *Transformer*-Objekt erzeugt (siehe Zeile 25-27), um dementsprechend die XSL-Transformationen durchzuführen (siehe Zeile 30-32).

Die Zieldateien sollen erst neu erzeugt werden, wenn an der modellierten FSM eine Änderung stattgefunden hat. Dies wird mit einer *if*-Anweisung sichergestellt (siehe Zeile 22). Die *if*-Anweisung sorgt dafür, dass die Transformationsvorgänge erst ausgeführt werden, wenn die XML-Daten der entsprechenden FSM modifiziert wurden.

6. Evaluierung

In diesem Teil der Arbeit soll sichergestellt werden, dass das erstellte OMNeT++ FSM Graphical Editor (OFGE) Plugin gemäß der in Kapitel 3 spezifizierten Anforderungen funktioniert.

Da die OMNeT++ Simulation auf verschiedenen Betriebssystemen (Linux, Windows und Mac OS) genutzt wird, muss als erstes überprüft werden, ob das entwickelte OFGE Plugin auf den genannten Betriebssystemen erfolgreich und problemlos installierbar ist.

Nachdem das OFGE erfolgreich installiert wurde, muss geprüft werden, ob die Erstellung einer **.fsm*-Datei, die dem Editor entspricht, möglich ist (siehe *example.fsm* in Abbildung 6.1). Nach der Erstellung dieser Datei soll geprüft werden, ob die Werkzeuge (siehe Palette in Abbildung 6.1) zur Modellierung der FSMs zur Verfügung stehen.

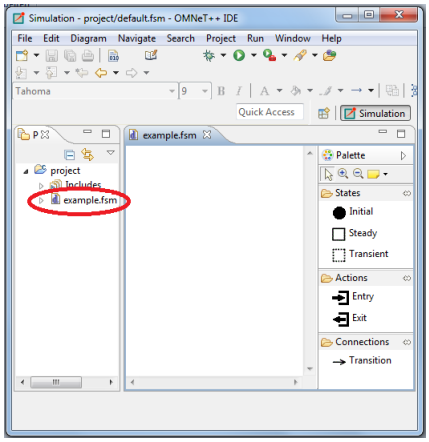

Abbildung 6.1.: Der OMNeT++ FSM Graphical Editor (OFGE) (*example.fsm*)

Während der Modellierung der FSM muss getestet werden, ob bestimmte Regeln, die bei der Implementierung definiert wurden, vom OFGE überprüft werden. Beispielsweise kann der Nutzer nicht mehr als einen Initial-Zustand zu einer FSM hinzufügen, da durch den OFGE definiert wurde, dass eine FSM nur einen Initial-Zustand besitzen darf.

Sobald die gewünschte FSM modelliert und gespeichert wurde, muss sicher gestellt werden, dass die entsprechende Dateien, die in Abschnitt 5.2 dargestellt wurden, generiert werden (siehe Abbildung 6.2).

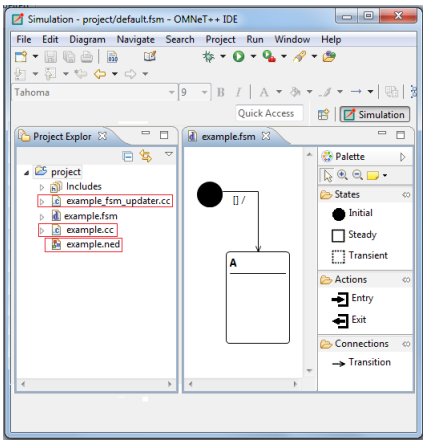

Abbildung 6.2.: OFGE generierte Dateien nach dem Speichern der modellierten FSM

Zusätzlich muss geprüft werden, ob der Inhalt der generierten Dateien mit der modellierten FSM übereinstimmt. Anschließend muss nach dem Start der OMNeT++ Simulation geprüft werden, ob die FSM zur Laufzeit richtig dargestellt wird und sich zum richtigen Zeitpunkt aktualisiert.

Folgende Tabelle zeigt alle getesteten Anforderungen. Das OFGE-Plugin konnte diese erfolgreich erfüllen. Somit ist die Evaluierung des OFGE-Plugins erfolgreich.

Anforderung	**Erfüllt?**
Wurde das OFGE-Plugin erfolgreich installiert?	Ja
Wurde die **.fsm*-Datei erfolgreich erstellt?	Ja
Konnte die gewünschte FSM in der **.fsm*-Datei modelliert werden?	Ja
Wurden nach Speicherung der **.fsm*-Datei die Dateien erzeugt?	Ja
Stimmt der Inhalt der Daten mit der modellierten FSM überein?	Ja
Wird die modellierte FSM zur Laufzeit richtig dargestellt?	Ja
Verhält sich die zur Laufzeit dargestellte FSM richtig?	Ja

Tabelle 6.1.: Testkriterien für OMNeT++ FSM Graphical Editor (OFGE) Plugin

7. Fazit

Thema der Arbeit war es, Finite State Machines (FSMs) in die OMNeT++ Simulation zu integrieren. Dem Nutzer soll ein graphischer Editor zur Verfügung gestellt werden, der zur Modellierung von FSMs für die OMNeT++ Komponenten dient. Aus den modellierten FSMs soll der Implementierungscode automatisch generiert werden. Zusätzlich sollen die modellierten FSMs zur Laufzeit dargestellt und aktualisiert werden.

Um eine einheitliche Wissensbasis zu schaffen, wurden zu Beginn der Arbeit einige Grundlagen vermittelt. Als erstes wurde die Simulationsumgebung OMNeT++ vorgestellt. Da OMNeT++ in das Eclipse Framework eingebettet ist und daraus resultierend Eclipse Plugins am besten geeignet sind, um eine Erweiterung in die vorhandene Plattform zu integrieren, wurden die Grundlagen für Eclipse Plugins kurz erläutert. Danach wurden Grundlagen zur Codegenerierung erläutert, da im Rahmen dieser Arbeit aus den modellierten FSMs Code generiert wird. Anschließend wurden Grundlagen zu den Finite State Machines vorgestellt. Da in dieser Arbeit die OMNeT++ FSM als Ansatz für die Implementierung der FSMs eingesetzt wird, wurde am Ende die OMNeT++ Finite State Machine ausführlich dargestellt.

Im Hauptteil der Arbeit wurden Anforderungen an das Programm gestellt. Anhand der gestellten Anforderungen an die verschiedenen Ebenen des Programms konnte das Konzept erstellt werden, welches daraufhin erfolgreich implementiert worden konnte. Aufgrund der darauffolgenden Evaluierung kann anschließend gesagt werden, dass das Ziel der Arbeit erreicht wurde. Über das entwickelte Plugin ist der Nutzer in der Lage, FSMs für OMNeT++ Komponenten mittels eines graphischen Editors (OFGE) zu modellieren. Aus den modellierten FSMs wird Implementierungscode automatisch generiert und anschließend werden die FSMs zur Laufzeit in der OMNeT++ Simulation dargestellt. Somit kann das Verhalten der einzelnen OMNeT++ Komponenten zur Laufzeit beobachtet bzw. überprüft werden, um beispielsweise Systemfehler aufdecken zu können.

OFGE-Manual

Im Anhang befindet sich das OFGE-Manual. Es enthält eine kurze Einführung, eine Installationsanleitung und ein Tutorial, um die einzelnen Schritte zum Aufbau einer FSM mit Hilfe des OFGE zu erläutern.

OMNeT++ FSM Graphical Editor (OFGE)

User Manual

Nebal El Bebbili
HAW Hamburg
CoRE Research Group

Contents

1 Introduction

OMNeT++ already provides Finite State Machines (FSMs). To know how these work and are structured, please read the „Finite State Machines“ section of the OMNeT++ manual. The german documentaion of the OMNeT++ FSMs can be found under the section 2.4.1 of the thesis documentation. The „Finite State Machines“ section of this munual illustrates that the user has to deal with many lines of code to build an OMNeT++ FSM. To make this procces easier for the user, the OFGE was designed. It provides the following features:

- The user will be able to build a FSM using a graphical editor. The required code will be automatically generated.

- This FSM will be represented during runtime. This allows the user to observe the FSM dynamically in order to know: The current state of the designed FSM at a certain time and which transition has been used to get into the current state.

2 Installation

This section shows step by step how to integrate the plugin into your OMNeT++ framework.

2.1 Installation Conditions

- The OFGE plugin works only on OMNeT++ 5.0 or higher (tested under OMNeT++ 5.0b1).
- The required Java version under Eclipse must be 1.8 or higher. To see the one Eclipse is running under on Windows or Linux, go to *Help->About OMNeT++ IDE->Installation Details->Configuration*, look for the property *eclipse.vm*. On Mac/OS go to *OMNeT++ IDE->About OMNeT++ IDE->Installation Details->Configuration*.

  For example: ***eclipse.vm=C:/Program Files (x86)/java/jre1.8.0_60/bin/javaw.exe***

  If the Java version is lower you can change the VM in the omnetpp.ini file under *<yourPath>OMNeTpp-„version“/ide/omnetpp.ini* by entering following two lines under the line „OMNeT++ IDE“.

  Windows
  -vm
  C:\jdk1.8.0\bin\javaw.exe (your exact path to javaw.exe could be different, of course)

  Linux
  -vm
  /opt/sun-jdk-1.8.0/bin/java (your exact path to javaw.exe could be different, of course)

Mac/OS
-vm
/Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/bin/java (your exact path to javaw.exe could be different, of course)

2.2 Installing OFGE

The OFGE plugin is stored in the directory „*application\org.omnetpp.core.fsm.site*“ of the tar-archive. Copy it to the chosen path, where you want the plugin to be stored, for example: „*C:\org.omnetpp.core.fsm.site*“, then follow the steps bellow:

1 **Step:** Start OMNeT++ and select the menu item *Help* > *Instal New Software...* (see Figure 2.1)

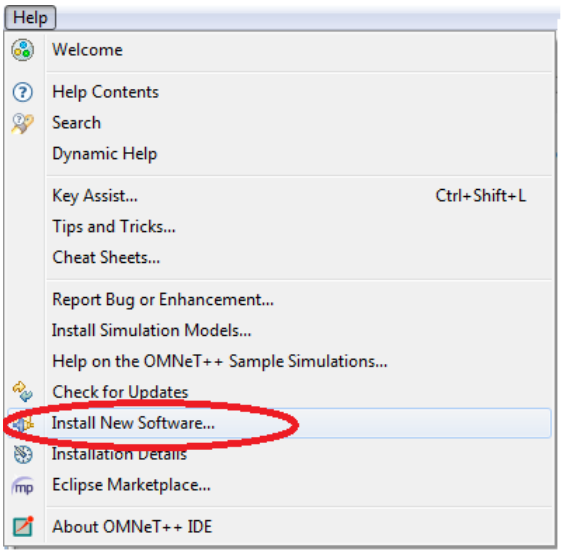

Figure 2.1: Selecting *Install New Software* under the menu item *Help*

2 **Step:** You can directly add the update site of OFGE by clicking the *Add* button (see Figure 2.2).

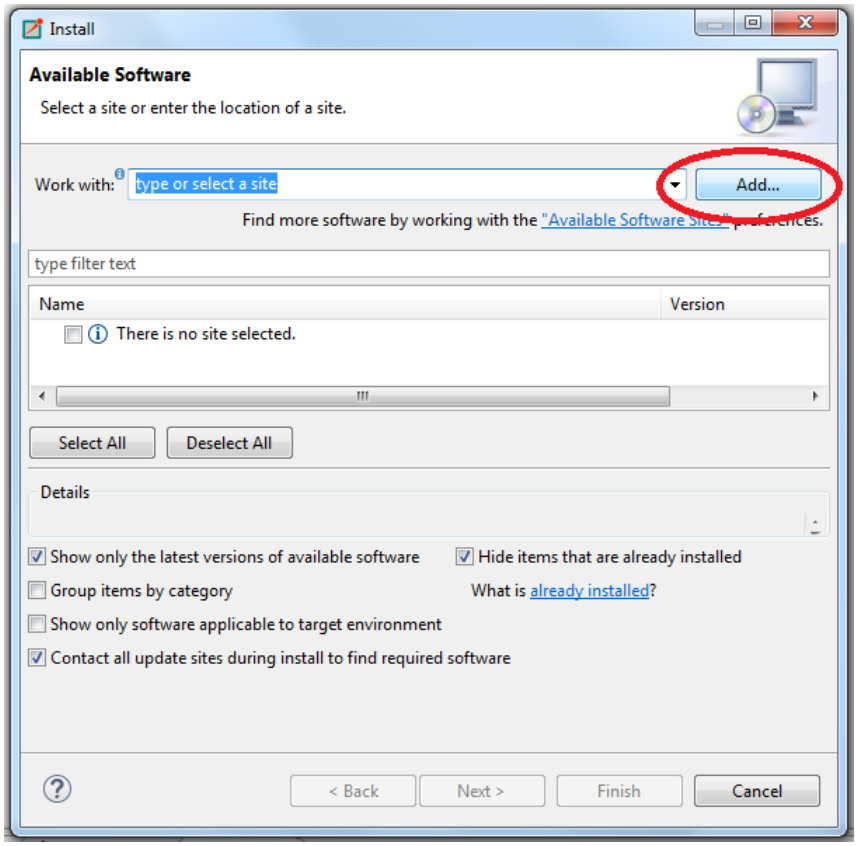

Figure 2.2: Install wizard of the Eclipse environment

3 **Step:** On the *Add Repository* window, click the *Local...* button and select the path of the *org.omnetpp.core.fsm.site* directory then click the *OK* button (see Figure 2.3).

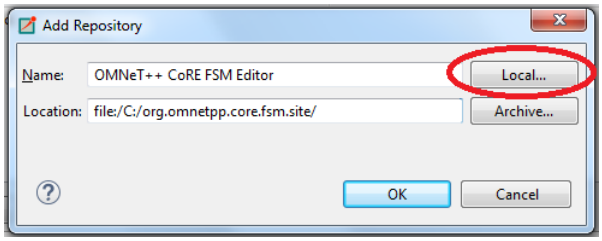

Figure 2.3: Selecting the site of OMNeT++ CoRE FSM Editor

4 **Step:** Now select OMNeT++ CoRE FSM Editor and click the *Next>* button (see Figure 2.4).

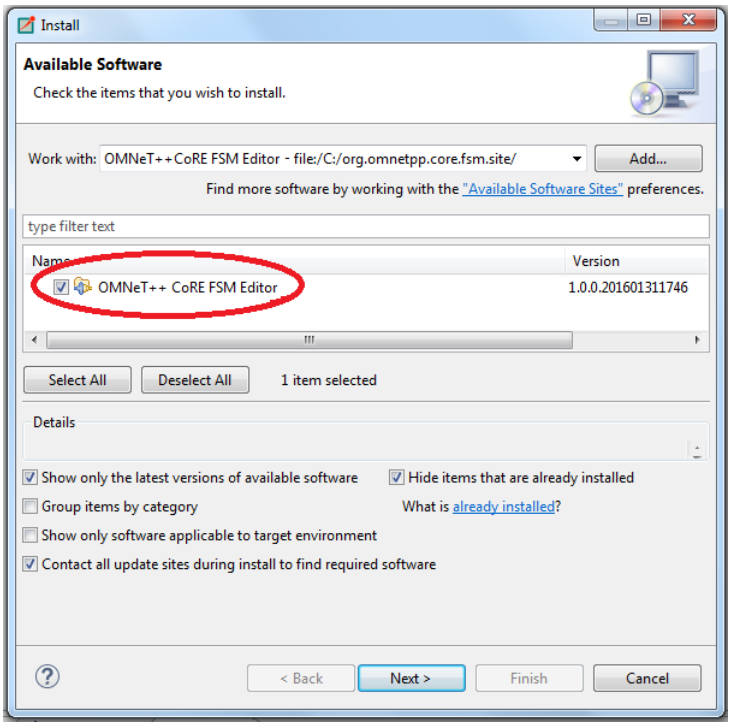

Figure 2.4: Selecting the OMNeT++ CoRE FSM Editor plugin to install it

5 **Step:** Click again the *Next>* button. Accept terms of license agreement and click the *Finish* button in order to start the download of OFGE. To apply installation changes and restart OMNeT++ click on the *Yes* button.

3 Using OFGE

3.1 Before you start

This section provides the information that you need to know before you build a FSM using OFGE.

3.1.1 Create FSM file

The OFGE plugin lets you create a FSM file **.fsm*, which is the editor where the FSM will be built:

1 **Step:** After the OFGE has been installed, choose *File* > *New* > *Other* >.. (see Figure 3.1).

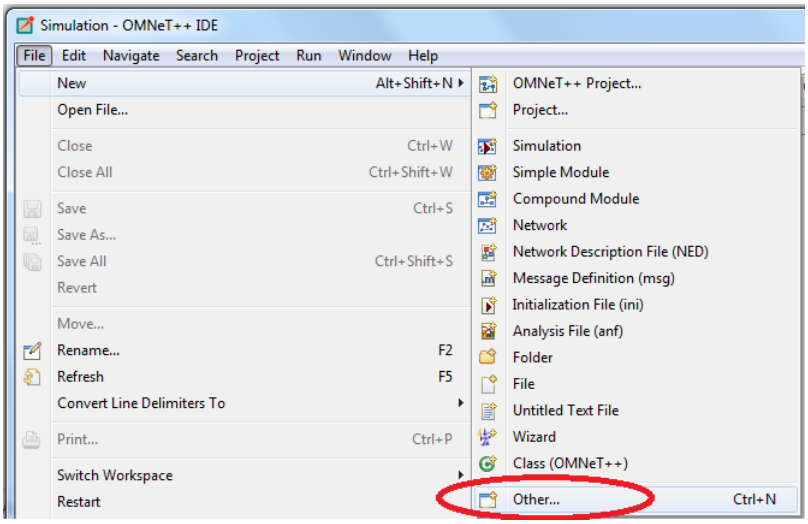

Figure 3.1: Create a FSM file by choosing *File* > *New* > *Other* >..

2 **Step:** After the *Select a Wizard* window shows up, select *OMNeT++ CoRE FSM Editor > Fsm Diagram* and click the *Next>* button (see Figure 3.2). Enter a name and save the file in the folder where you want it to be created and then click the *Finish* button (see Figure 3.3). The FSM file **.fsm* cannot have the name of a NED (Network Description Language) keyword like (default, simple, network etc.), this would cause an error in the generated **.ned* file (condition 1 under the chapter Conditions).

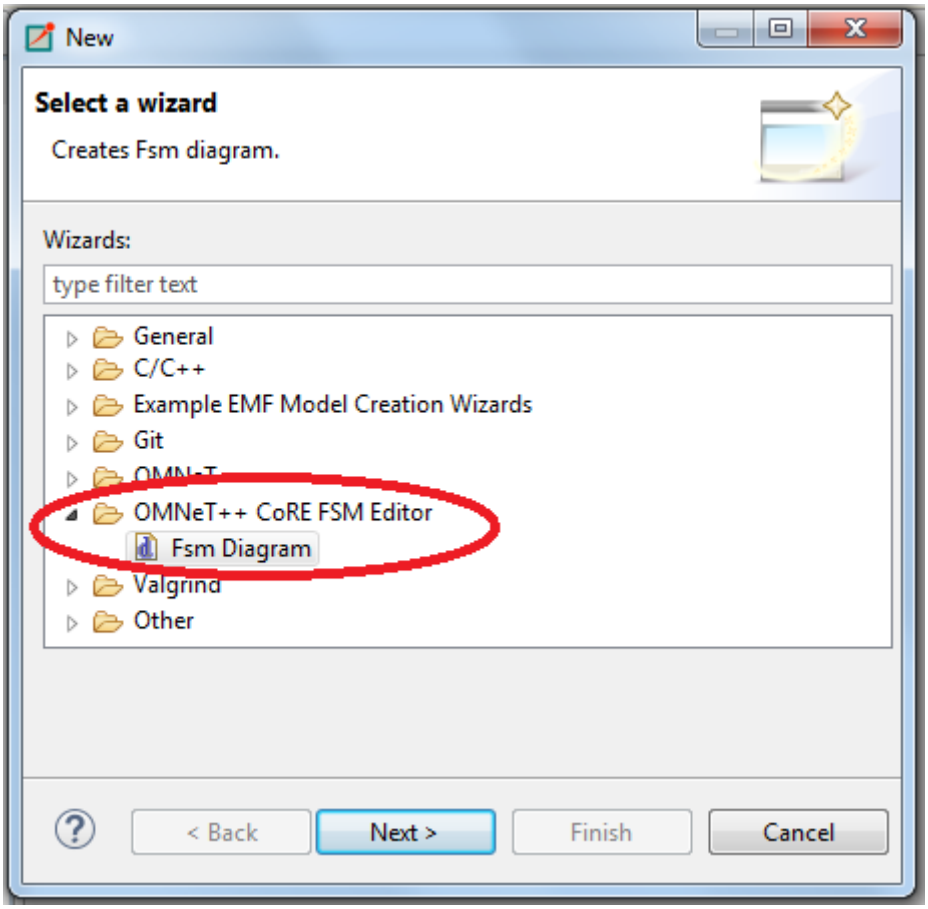

Figure 3.2: Creating a FSM file by choosing *OMNeT++ CoRE FSM Editor > Fsm Diagram* in the *Select a Wizard* window

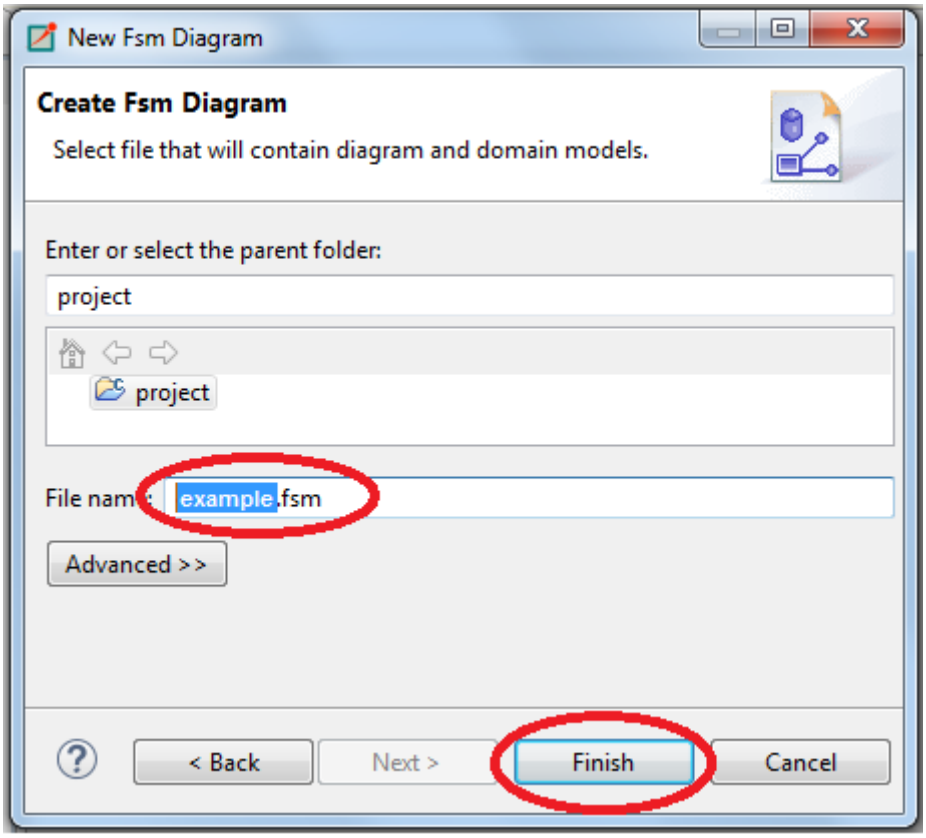

Figure 3.3: Giving the FSM file a name in the *New Fsm Diagram* window

As shown in Figure 3.4 the OFGE file „example.fsm“ is now created, where you can built your FSM.

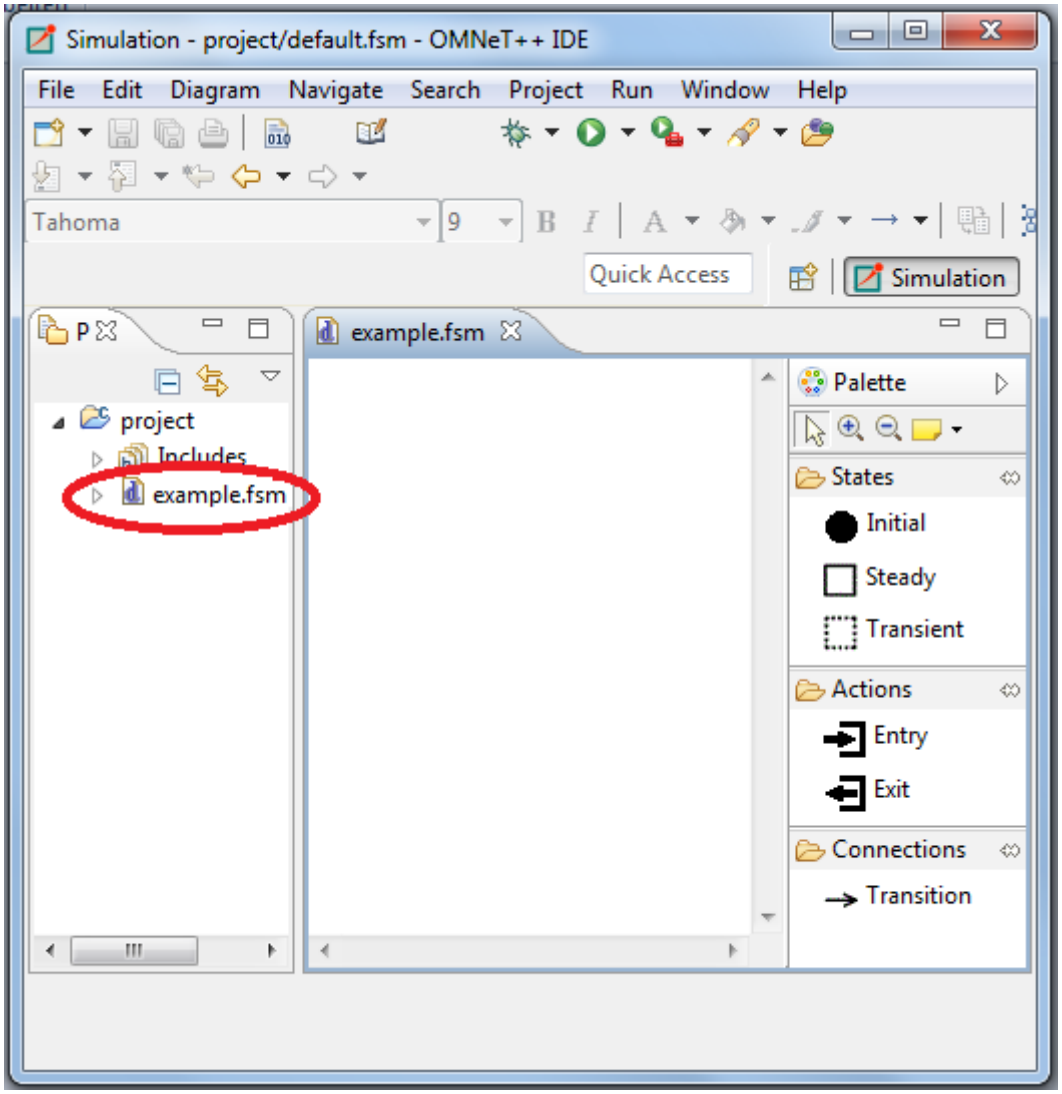

Figure 3.4: OFGE file (*.fsm)

3.1.2 Palette

The component palette at the right side of the OFGE (*example.fsm*) contains all components, to build an FSM (see Figure 3.5). You can add these components to your FSM, by selecting a specific component from the palette and placing it in the FSM Diagram with a mouse click.

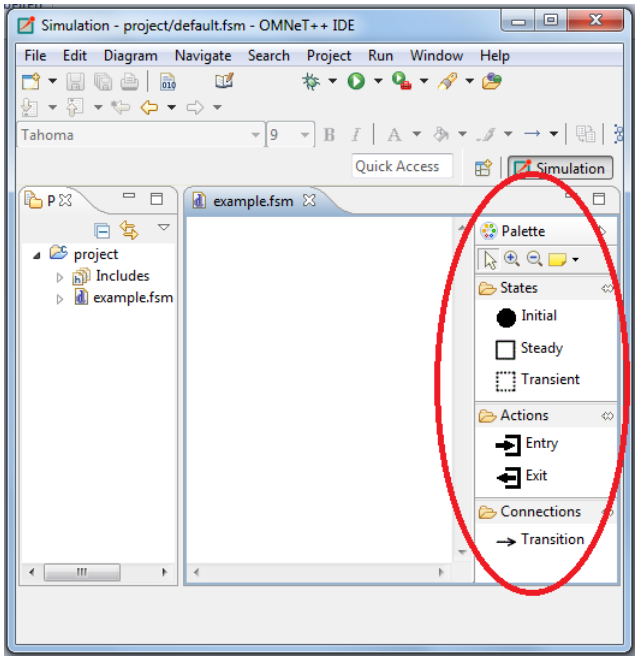

Figure 3.5: Palette of the OMNeT++ FSM-Editor

The rest of this section defines the graphical components of a FSM in detail.

States

As shown in Figure 3.5 and Figure 3.6 three kinds of states that exist:

- **Initial state:**
  According to UML state diagrams the black dot represents the initial state of the FSM (see. Figure 3.6). An initial state can point on several states. An FSM must always have an initial state to start at after reseting the simulation (condition 2 under the chapter 5 Conditions).

- **Transient and steady state:**
  As shown in Figure 3.6, steady and transient states have different layouts. A transient state is a rectangle with dashed lines, while a steady state has normal lines. Transient/steady states are resizable. Steady/transient states have the default name „state name“. You can change it by clicking on the „state name“ label or using the Properties view. A state name must not contain special characters (condition 3 under the chapter 5 Conditions).

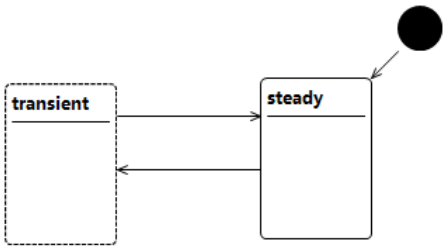

Figure 3.6: Layout of initial, steady and transient state

Transitions

Edges show the transitions from one state to another. Each transition is labeled. The label has two parts separated by a slash. The first is the Guard that triggers the transition. The second is the Effect, which will be performed once the transition has been triggered. (see Figure 3.7).

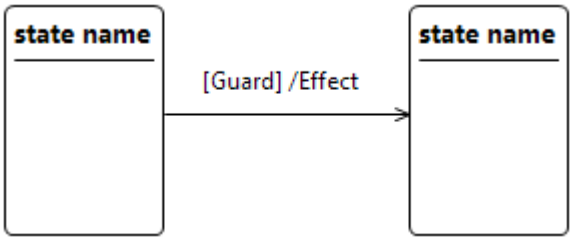

Figure 3.7: Transition layout

Example:

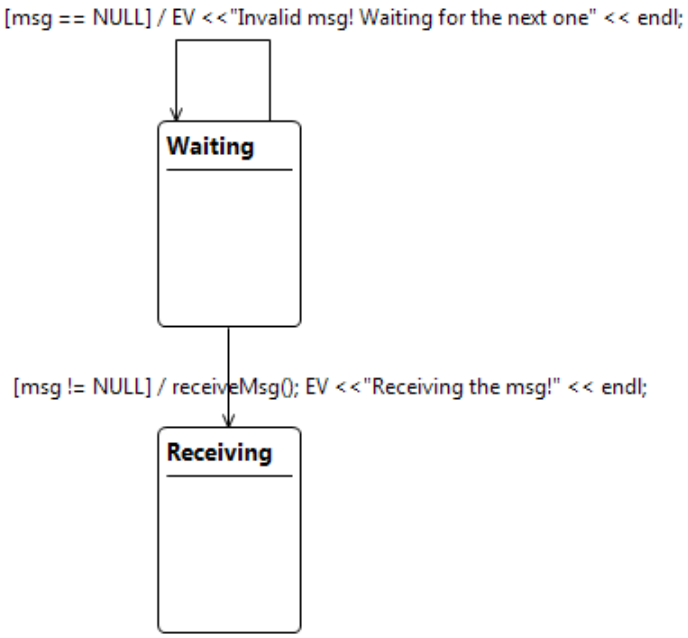

Figure 3.8: Transition example

We can interpret Figure 3.8 as follows:

- If the FSM is in the „Waiting“ state and an invalid msg arrives, then the FSM stays in the „Waiting“ state and the following Effect action is performed:

```
1        EV <<"Invalid msg! Waiting for the next one" << endl;
```

- If the FSM is in the „Waiting“ state and a valid msg arrives, then the FSM transitions to the „Receiving“ state, and the following Effect actions are performed.

```
1        receiveMsg();
2        EV <<"Receiving the msg!" << endl;
```

Guards and effects are also accessible from the Properties View.

An effect of a transition has to end with a semilcolon. If there are multiple effects, they must be separated by semicolons (condition 4 under the chapter 5 Conditions).

If you want to reroute a transition to a new source state after you've already connected it to its source and target state, you need to delete the transition and draw it again (rerouting the target state of a transition doesnot cause any problems) (condition 5 under the chapter 5 Conditions).

The guards of the transitions must be mutually exclusive. If this is not the case then, if a state has several transitions that can be switched, the first transition, which has the user drawn, will be executed (switched) (condition 6 under the chapter 5 Conditions).

Please take note of the condition 7 under the chapter Conditions.

Entry/Exit Actions

The lower area of transient and steady states is for their actions. Actions are divided in:

- Entry code: Actions that are executed when entering the state.
- Exit code: Actions that are executed when leaving the state.

These entry and exit components are under the category „Actions“ of the palette and can only be placed in the lower area of transient/steady states (see Figure 3.9). Each of these components is labeled, so you can add your code to the label. You can change the value of the label by clicking on it or using the Properties View. Entry and exit labels can have multiple constructions separated with semicolon (condition 8 under the chapter 5 Conditions).

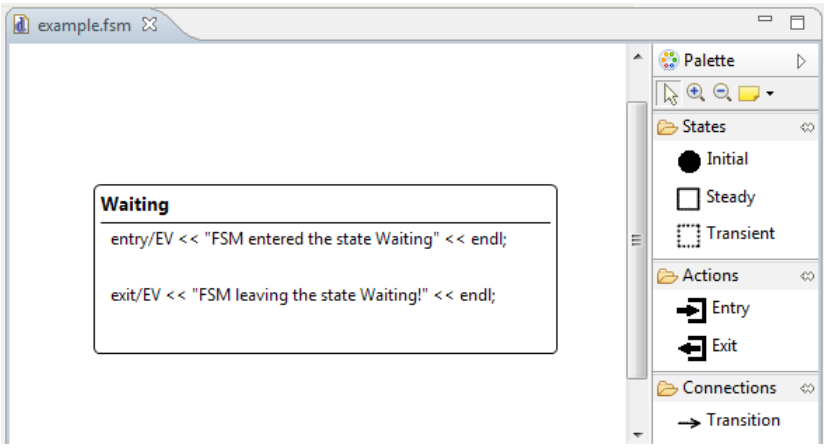

Figure 3.9: Entry-/exit-code example

Note

The Properties View can be used to edit the attributes of the FSM (state name, guard, effect, etc.). If you do not see the Properties View, go to *Window > Show View > General > Properties*.

3.1.3 Generated files

After saving the graphical representation of the FSM (**.fsm* file), four files will be generated:

1. **The *file *.cc***

   This file contains the C++ implemenation code of the OMNeT++ FSM that is modelled using OFGE. It includes the *FSM_Switch()* and thus the definition of all possible states and their behavior. This file has the same name as the **.fsm* file.

2. **The file **additional_code.h***

   This is an empty file where the user can add additional code such as variables and methods used by the FSM (by the **.cc* file). The **_additional_code.h* file is automatically included in the FSM (**.cc* file), so that none of the added code will be lost after the **.fsm* file has been resaved and the files have been regenerated. Eeach time the user saves the **.fsm* file, the four files (**.cc*, **_fsm_updater.cc*, **.ned* and **_additional_code.h*) will be regenerated and explicitly added code will disappear (condition 9 under the chapter 5 Conditions). This file has the name of the **.fsm* file extended with the string „*_additional_code*“.

3. **The file **_fsm_updater.cc***

   This file contains the module, which is responsible for the representation of the FSM and its updates during runtime. It has the name of the **.fsm* file extended with the string „*_fsm_updater*“.

4. **The file **.ned***

   This **.ned* file defines the previous two modules and an additional compound module, which contains the two sub modules (**.cc* and **_fsm_updater.cc*) and their connections. It also has the same name as the **.fsm* file.

The next section takes a closer look at these files.

3.2 Tutorial

This short tutorial to OMNeT++ FSM Graphical Editor (OFGE) guides you through an example. The example is based on the OMNeT++ Tictoc simulation tutorial with additionally funktions to show the features of the OFGE.

At the beginning we will look at the „Tic“ FSM, after that we take a quick look at the individual generated files. Finally, we start the simulation to see how the represented FSM during runtime looks like and what information it provides.

This tutorial consists of two modules „Tic“ and „Tic“. The modules will do something simple: Tic will create a packet, and the two modules will keep passing the same packet back and forth. Eeach of the modules has an FSM, in this tutorial we will only observe the FSM of the „Tic“ module.

The project can be found under the folder *application/omnetpp.fsm.editor.tutorial* of the tar-archive.

3.2.1 The FSM Editor **.fsm* file

As shown in Figure 3.10, the FSM „Tic“ has the following states:

- *INIT* state (initial state)
- *WAITING_TO_SEND* (steady state)
- *WAITING_TO_RECV* (steady state)
- *SEND* (transient state)
- *RECV* (transient state)

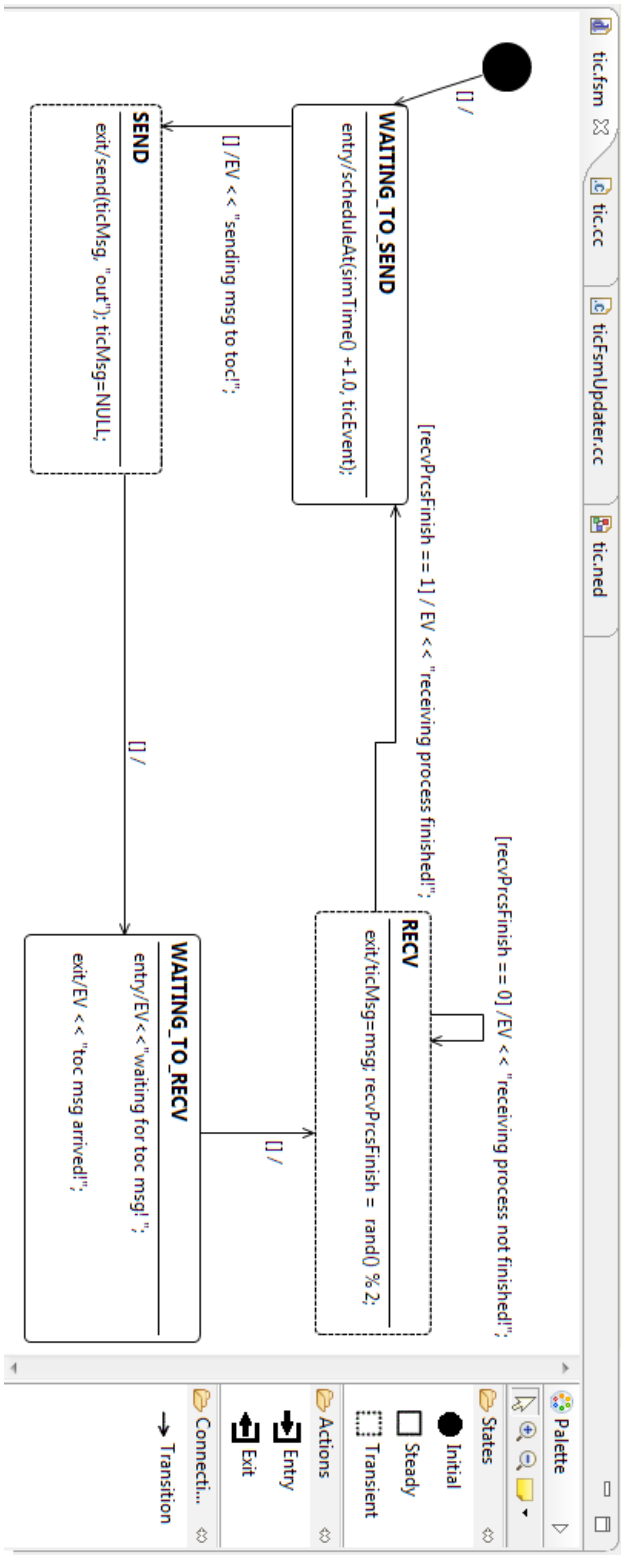

Figure 3.10: Tic FSM

Tic-FSM Process

The FSM will always start with initial state then switch to *WAIT_TO_SEND*. Upon entering this state, a timer will be set to start sending the msg to Toc. When the timer expires, the FSM will switch to the transient state *SEND*, then send the msg to Toc and switch again to the steady state *WAIT_TO_RECV*. Now the FSM will wait untill Toc sends the msg back. After the Toc msg arrives, the FSM switches to the transient state *RECV* and stays in it until the receiving process is finished (This is realized with the random variable *recvPrcsFinish* that can have the value 0 or 1). Once the receiving process is finished, the FSM switches to *WAIT_TO_SEND* and repeats the process.

3.2.2 The generated *.cc file

After saving the tic.fsm file of the OFGE, the tic.cc file will be generated. If you take a look at it, you will find:

- Definition of the tic class
- Definition of the Enum, which defines the states of tic
- Constructor
- Destructor
- Initialize method
- *HandleMessage* method, which contains the *FSM_Switch()*
- *SendUpdateMsg* and *sendTransitionUpdateMsg* methods, to update the represented FSM during runtime

FSM_Switch

Now if we take a closer look to the *FSM_Switch()*, we notice:

- There is a case statement for each entry and exit label of the FSM (see Figure 3.11).
- There is an if-statement for each transition of the FSM (see Figure 3.12):
  . The condition (boolean expression) of the if-statement corresponds to the guard of the transition in the FSM.
  . The code-block of the if-statement corresponds to the effect of the transtion in the FSM.
  . The target state of the transition corresponds to the second parameter of the invoked method *FSM_Goto()*.

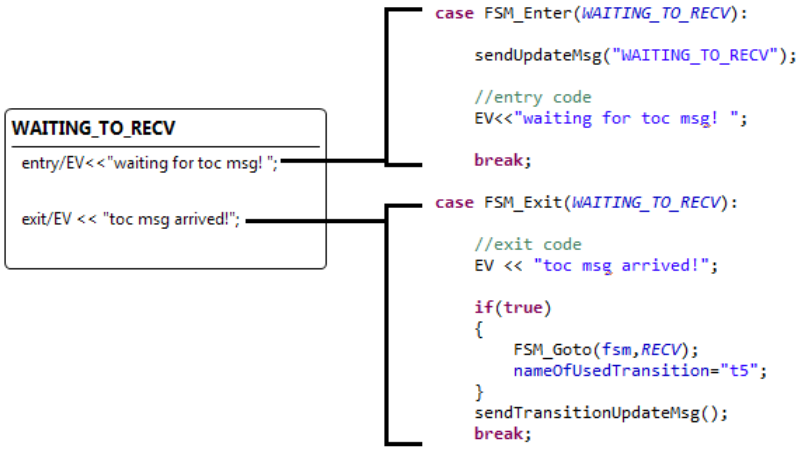

Figure 3.11: generated code example of entry and exit labels

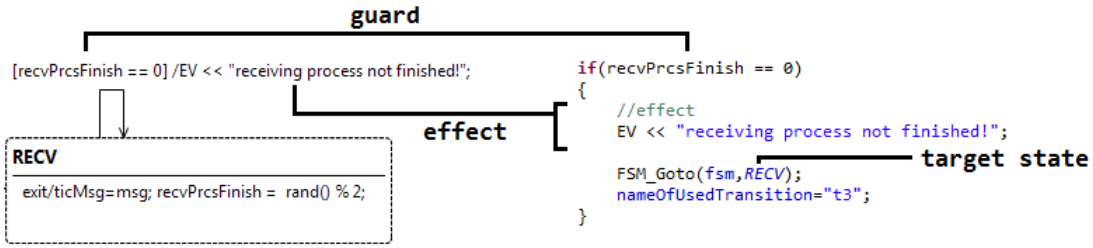

Figure 3.12: generated code example of a transition

3.2.3 The generated *_additional_code.h file

As already mentioned in section 3.1.3, the *_additional_code.h file is where the user can add additional code used by the FSM like methods and variables, so the added code will not disappear after regenerating the *tic.cc* file. In this tutorial the following code sections of the tic.cc module needs to be modified:

- The class definition
- The constucture and destructure
- The *initialize()* method (see red marked lines in Figure 3.13)

To make sure that the modification won't be lost after regenerating the *tic.cc* file, the modified code will be stored in the *tic_additional_code.h* (see Figure 3.13)

```
7  class tic: public cSimpleModule{
8      cFSM fsm;
9      enum {
10           INIT = 0,
11           //state name = FSM_type(position)
12           WAITING_TO_RECV = FSM_Steady(1),
13           WAITING_TO_SEND = FSM_Steady(2),
14           SEND = FSM_Transient(1),
15           RECV = FSM_Transient(2),
16           };
17
18     //this msg is used to update the online FSM
19     cMessage* updateMsg;
20     String nameOfUsedTransition;
21
22     cMessage *ticEvent, *ticMsg;
23     int recvPrcsFinish;
24
25 public:
26     tic();
27     virtual ~ tic();
28
29 protected:
30     virtual void initialize();
31     virtual void handleMessage(cMessage *msg);
32     virtual void sendUpdateMsg(char* nameOfCurrentState);
33     virtual void sendTransitionUpdateMsg();
34 };
35
36 tic:: tic() {
37     ticEvent = ticMsg = NULL;
38 }
39
40 tic::~ tic() {
41     delete ticMsg, ticEvent;
42 }
43
44 void  tic::initialize() {
45     updateMsg = new cMessage();
46     fsm.setName("fsm");
47     ticEvent = new cMessage("ticEvent");
48     ticMsg = new cMessage("ticMsg");
49     //Scheduling first send
50     scheduleAt(5.0, ticEvent);
51 }
```

Figure 3.13: added code lines in *tic.cc*

3.2.4 The generated *_fsm_updater.cc file

This file is responsible for the representation of the FSM during runtime. You don't need to change or add code to this file, unless you want to change the layout of the represented FSM during runtime.

3.2.5 The generated tic.ned file

The genereted NED file contains:

- Integration of the FSM updater module *tic_fsm_updater.cc*, which updates the FSM during runtime, it has an input gate that is connected to the main module *tic.cc* to receive the update signals.
- Integration of the main module *tic.cc*, with input and output gates.
- Integration of the compound module, which connects the previous two modules as submodules. The compound module also has parameters, which correspond to the figures of the represented FSM during runtime (states, transitions, etc.)
- Integration of the network that includes the compound module.

3.2.6 The represented FSM during runtime

After starting the simulation, click on the compound module of Tic to see the representation of FSM (see Figure 3.14).

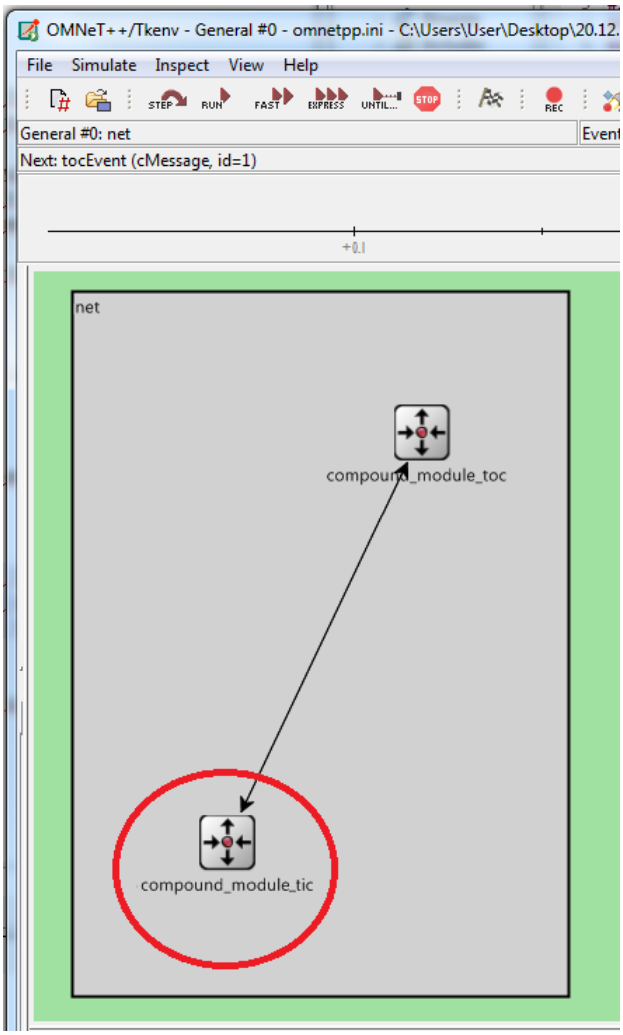

Figure 3.14: Tic and Toc compound modules

In Figure 3.15 we notice that the represented FSM during runtime has the same look as the FSM of OFGE, only that entry and exit codes are displayed under the FSM diagram instead of isnide the states. In addition, this FSM is only read and it can update its layout by:

- updating the color of the current state by changing it into red.
- updating the color of the used trasition by changing it into bule.

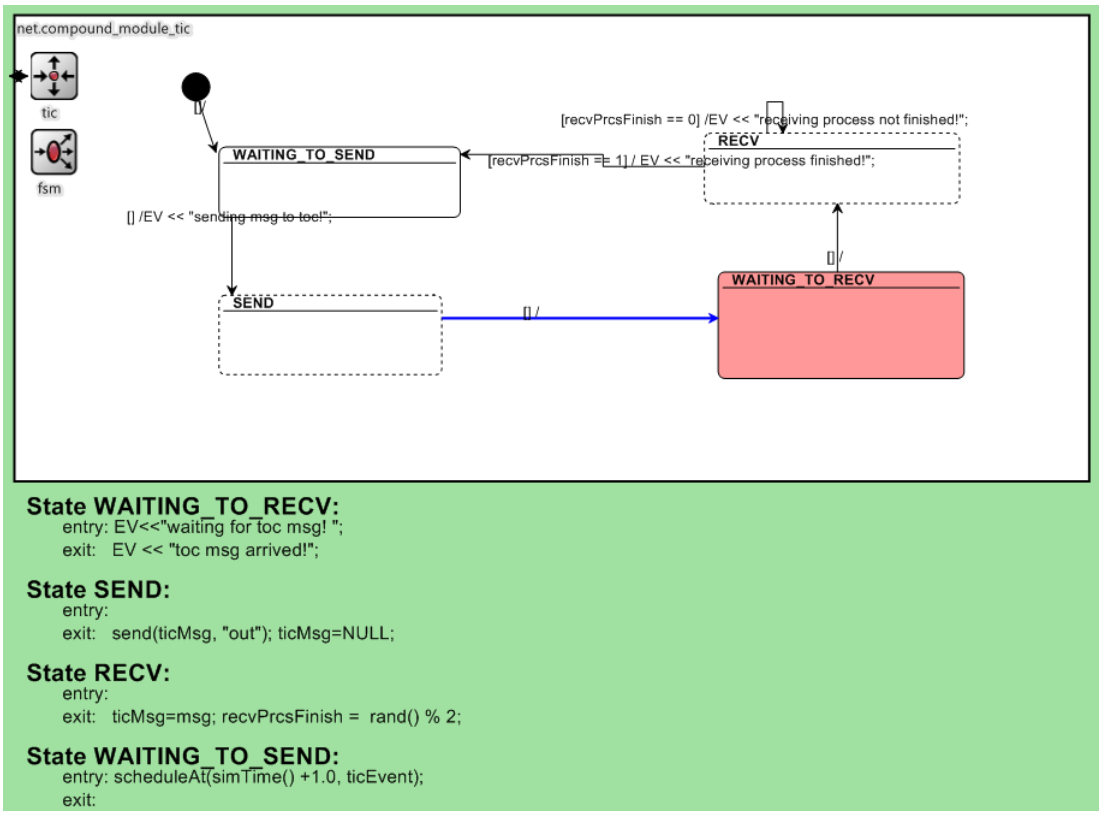

Figure 3.15: The represented FSM after starting the simulation

4 Conditions

1. The FSM file **.fsm* cannot have the name of a NED (Network Description Language) keyword like (*default*, *simple*, *network* etc.), this would cause an error in the generated **.ned* file.
2. An FSM must always have an initial state to start at after reseting the simulation.
3. A state name must not contain special characters.
4. An effect of a transition has to end with a semilcolon. If there are multiple effects, they must be separated by semicolons.
5. If you want to reroute a transition to a new source state after you've already connected it to its source and target state, you need to delete the transition and draw it again (rerouting the target state of a transition doesnot cause any problems).
6. When a state has several transitions that can be switched, the first transition, which has the user drawn, will be executed (switched).

7. The OFGE ist based on GMF (Graphical Modeling Framework), that has a bug which leads to:

   In some cases, the connections (transitions) of the represented FSM during runtime do not look like the connections of the Editor FSM (see Figure 4.1).

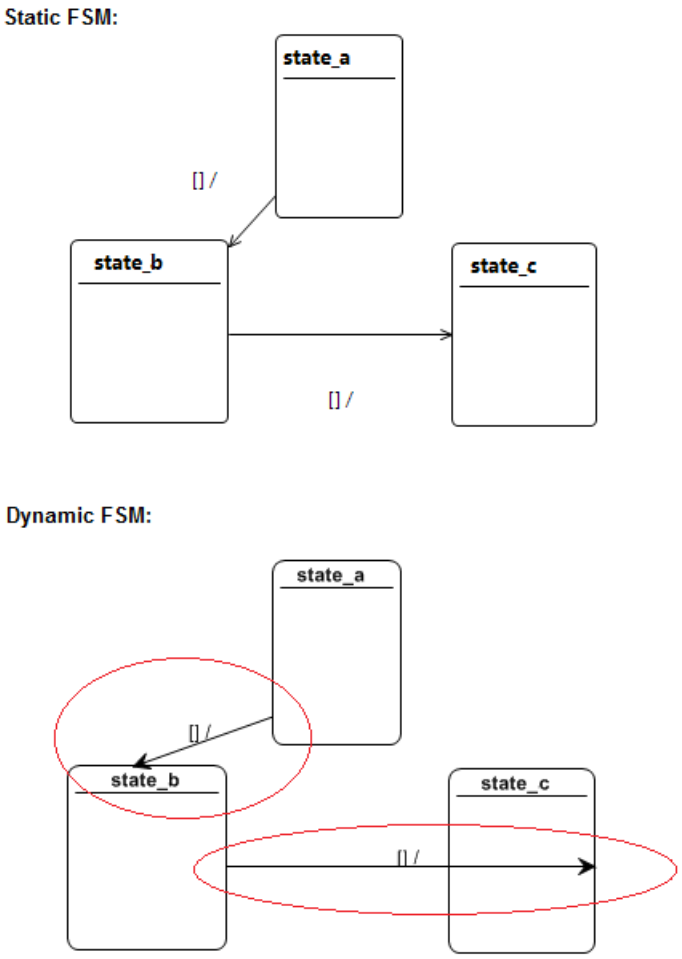

Figure 4.1: FSM of OFGE vs. represented FSM during runtime (connections don't match)

To avoid this, a new bendpoint needs to be added to a transition:

- After connecting the source/target states with a transtion.
- After moving source or target state.
- After changing the position of the source/target anchor within the same state.
- Even after rerouting the link to the new target.

Creating bendpoints

To create a new bendpoint (see Figure 4.2):

a) Hold the transition at the position you want to create the bendpoint

b) Move the cursor in any direction (in this case down)

c) The bendpoint is created

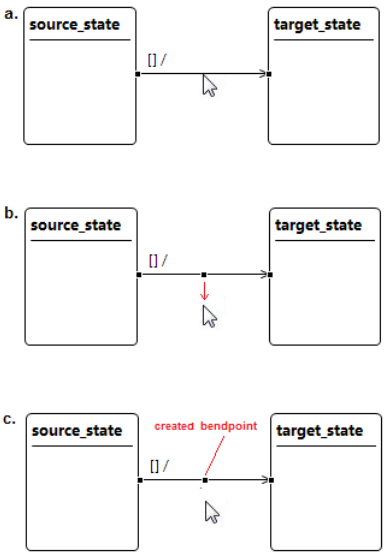

Figure 4.2: Creating a transition bendpoint

8. Entry and exit labels can have multiple constructions separated with semicolon.
9. Eeach time the user saves the **.fsm* file, the four files (**.cc*, **_fsm_updater.cc*, **.ned* and **_additional_code.h*) will be regenerated and explicitly added code will disappear.

Literaturverzeichnis

[Balzert 2005] Balzert, Heide: *Lehrbuch der Objektmodellierung.* Spektrum Akademischer Verlag, 2005

[Bongers 2004] Bongers, Frank: *XSLT 2.0. Das umfassende Handbuch.* Galileo Press, 2004. – ISBN 3-89842-361-1

[Brambilla u. a. 2012] Brambilla, Marco ; Cabot, Jordi ; Wimmer, Manuel: *Model-Driven Software Engineering in Practice.* Morgan & Claypool.verlag, 2012. – ISBN 9781608458820

[CoRE Research Group] CoRE Research Group. – http://core.informatik.haw-hamburg.de/en/ - Zugriffsdatum: 2015-10-22

[Eclipse Foundation a] Eclipse Foundation: *Eclipse IDE.* – URL http://www.eclipse.org/. – Zugriffsdatum: 2015-12-14

[Eclipse Foundation b] Eclipse Foundation: *Eclipse Modeling Framework (EMF).* – URL https://eclipse.org/modeling/emf/ - Zugriffsdatum: 2016-01-25

[Eclipse Foundation c] Eclipse Foundation: *Graphical Editing Framework (GEF).* – URL https://eclipse.org/gef/ - Zugriffsdatum: 2016-01-25

[Eclipse Foundation d] Eclipse Foundation: *Graphical Modeling Framework Tooling Version 3.2.1.* – URL http://www.eclipse.org/gmf-tooling/ - Zugriffsdatum: 2015-10-05

[Engelmann] Engelmann, Roman: *EMF - Eclipse Modeling Framework Seminararbeit im Seminar - Information Systems Engineering.* – URL http://docplayer.org/7942548-Emf-eclipse-modeling-framework-seminararbeit-im-seminar-html. – Zugriffsdatum: 2016-1-22

[Gamma, Erich u. a. 2015] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software.* mitp Verlags GmbH & Co. KG, 2015

[Hauer 2009–2010] Hauer, Philipp: *Das State Design Pattern.* 2009-2010. – http://www.philipphauer.de/study/se/design-pattern/state.php - Zugriffsdatum: 2016-01-08

[Hopcroft, John u. a. 2011] Hopcroft, John ; Motwani, Rajeev ; Ullman, Jeffrey: *Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit.* Pearson Studium, 2011

[Kecher 2011] Kecher, Christoph: *UML 2. Das umfassende Handbuch.* Galileo Press, 2011. – ISBN 978-3-8362-1752-1

[Object Management Group] Object Management Group: *Unified Modeling Language.* – http://www.uml.org/ - Zugriffsdatum: 2016-01-15

[OMNeT++ Community a] OMNeT++ Community: *OMNeT++ 5.0b1.* – URL http://www.omnetpp.org - Zugriffsdatum: 2015-10-05

[OMNeT++ Community b] OMNeT++ Community: *User Manual - OMNeT++ Version 4.6.* – URL https://omnetpp.org/doc/omnetpp/manual/usman.html - Zugriffsdatum: 2015-10-09

[Pietrek, Georg u. a. 2007] Pietrek, Georg ; Trompeter, Jens ; Beltran, Juan Carlos Flores ; Holzer, Boris ; Kamann, Thorsten ; Kloss, Michael ; Mork, Steffen A. ; Niehues, Benedikt ; Thoms, Karsten: *Modellgetriebene Softwareentwicklung.* entwickler.press, 2007. – ISBN 978-3-939084-11-2

[Prinz und Kirch 2013] Prinz, Peter ; Kirch, Ulla: *C. Lernen und professionell anwenden.* mitp, 2013. – ISBN 978-3-8266-9504-9

[Reussner und Hasselbring 2006] Reussner, Ralf ; Hasselbring, Wilhelm: *Handbuch der Software-Architektur.* 1. Heidelberg : dpunkt.verlag, 2006. – ISBN 3-89864-372-7

[Schäuffele und Zurawka 2006] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering - Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen.* 3. akt. u. verb. Aufl. 2006. Wiesbaden : Vieweg+Teubner Verlag, 2006. – ISBN 978-3-834-80051-0

[Scheibl] Scheibl, Hans-Jürgen: *Unified Modeling Language (UML).* – http://home.f1.htw-berlin.de/scheibl/uml/index.htm?./Zustandsautomat/Grundlagen.htm - Zugriffsdatum: 2016-01-07

[Shavor, Sherry u. a. 2004] Shavor, Sherry ; Fairbrother, Scott ; D'Anjou, Jim ; Kellerman, John ; Kehn, Dan ; McCarthy, Pat: *Eclipse. Anwendungen und Plug-Ins mit Java entwickeln.* Addison-Wesley Verlag, 2004. – ISBN 3-8273-2125-5

[Stahl, Thomas u. a. 2007] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung.* Heidelberg : dpunkt.verlag GmbH, 2007

[Todorov Todorov 2013] Todorov Todorov, Lazar: *Integration des AS6802 Synchronisationsprotokolls in eine OMNeT++ basierte Simulationsumgebung*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2013

[Varga 2014] Varga, Andras: *OMNeT++ Community Summit. Canvas API in OMNeT++.* 2014. – URL https://www.youtube.com/watch?v=yTsrXblBOVg

[Varga und Hornig 2008] Varga, András ; Hornig, Rudolf: An overview of the OMNeT++ simulation environment. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, networks and systems & workshops.* New York : ACM-DL, März 2008, S. 60:1–60:10. – URL http://portal.acm.org/citation.cfm?id=1416222.1416290. – ISBN 978-963-9799-20-2

[Vogel 2015] Vogel, Lars: *Eclipse Modeling Framework (EMF) - Tutorial.* 2015. – URL http://www.vogella.com/tutorials/EclipseEMF/article.html#emfeditor_generate. – Zugriffsdatum: 2016-1-21

[World Wide Web Consortium 2007] World Wide Web Consortium: *XSL Transformations (XSLT) Version 2.0.* 2007. – http://www.w3.org/TR/xslt20/ - Zugriffsdatum: 2015-10-23

Abkürzungsverzeichnis

API Application Programming Interface

CM Compression Master 23

CoRE Communication over Realtime Ethernet 1

DSL Domain Specific Language 28

EMF Eclipse Modeling Framework 28

FSM Finite State Machine 1

GEF Graphical Editing Framework 28

GMF Graphical Modeling Framework 8

HAW Hochschule für angewandte Wissenschaften

NED Network Description 3

OFGE OMNeT++ FSM Graphical Editor v

OMNeT++ Objective Modular Network Testbed in C++ 3

PHP Hypertext Preprocessor

SC Synchronisation Client 23

SM Synchronisation Master 23

UML Unified Modeling Language 9

XML Extensible Markup Language

XSLT Extensible Stylesheet Language Transformation 8

XSL Extensible Stylesheet Language

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 4. Mai 2016 Nebal El Bebbili