Fakultät II - Departement für Informatik

Individuelles Projekt
Entwicklung eines Javatools zur Übersetzung von Fehlerbäumen mit DC Semantik in Phasenautomaten

vorgelegt von: Casjen Schnars
Erstprüfer: Prof. Dr. Ernst-Rüdiger Olderog
Zweitprüfer: Dipl.-Inform. Andreas Schäfer

Oldenburg, den 22. Dezember 2004

Danksagung

Ich möchte mich hier bei allen Menschen bedanken, die es mir möglich gemacht haben dieses Individuelle Projekt erfolgreich durchzuführen.
Mein Dank geht an Dipl. Inf. Andreas Schäfer für seine kompetente und konsequente Betreuung während und auch über seine Arbeitszeit hinaus.
Für die Hilfe beim Umgang mit MobyDC, CVS und ant danke ich Dipl. Inf. Michael Möller.
Für die Betreuung während des Individuellen Projektes danke ich ebenfalls Prof. Dr. Ernst-Rüdiger Olderog.
Mein ganz besonderer Dank geht an meine Lebensgefährtin Janna Arnold für ihre Unterstützung während der gesamten Zeit des Individuellen Projektes, ihr Verständnis für die vielen Stunden am Rechner und ihre Geduld beim Zuhören.

Zusammenfassung

Mit Hilfe der Fehlerbaumanalyse wird die Sicherheit von technischen Systemen untersucht. Man kann für Fehlerbäume eine Semantik in der temporalen Logik Duration Calculus definieren. Anschließend können diese Fehlerbäume mit DC Semantik mit Hilfe einer existierenden Abbildungsvorschrift in Phasenautomaten umgewandelt werden. Die so entstandenen Automaten lassen mit dem Model-Checking Tool MobyDC eine Überprüfung der Fehlerbäume auf Korrektheit und Vollständigkeit zu.
In dieser Arbeit soll ein Java Tool erstellt werden, mit dem es möglich ist Fehlerbäume zu zeichnen und mit Hilfe des Model-Checkers MobyDC automatisch zu überprüfen, ob der Fehlerbaum korrekt und vollständig ist. Weiterhin soll der Umgang mit dem Tool anhand eines Fallbeispiels demonstriert werden.

Inhaltsverzeichnis

1 Einleitung **9**
1.1 Ziel des Projektes . . . 9

2 Grundlagen **10**
2.1 Duration Calculus with Liveness(DCL) . . . 10
2.2 Fehlerbaumanalyse . . . 12
2.2.1 Intention . . . 12
2.2.2 Gattertypen und Events . . . 12
2.2.3 Beispiel . . . 13
2.2.4 DCL-Syntax in Faulttree2Phaseautomata . . . 14
2.3 Phasenautomaten . . . 14
2.3.1 Definition der Syntax . . . 15
2.3.2 Uhren in Phasenautomaten . . . 15
2.3.3 Beispiel . . . 16
2.4 MobyDC . . . 16
2.5 Model-Checking in MobyDC . . . 16
2.5.1 Pattern 1 - erreichbarer Zustand . . . 17
2.5.2 Pattern 2 - Endzustand . . . 18
2.5.3 Pattern 3 - Sequenz . . . 18

3 Umsetzung **20**
3.1 Vorüberlegung . . . 20
3.1.1 Vorgehensweise . . . 20
3.1.2 Namensgebung . . . 20
3.2 Verwendete Techniken . . . 20
3.2.1 Java . . . 20
3.2.2 JGraph . . . 21
3.2.3 JFlex . . . 21
3.3 Oberflächendesign . . . 21
3.3.1 Textmenü . . . 23
3.3.2 Obere Toolbar . . . 24
3.3.3 Linke Toolbar . . . 26
3.3.4 Editieren eines Events . . . 26
3.4 Programm-Architektur . . . 26
3.4.1 Entwurf der Oberfläche . . . 28
3.4.2 Entwurf der Logik . . . 30
3.4.3 Export der Fehlerbäume in Phasenautomaten . . . 35
3.4.4 Datenhaltung . . . 36
3.4.5 XML-Datenformat . . . 36

4 Fallbeispiel **40**
4.1 Beschreibung des Systems 40
4.2 Fehlerbaum-Analyse 41
4.3 Modellierung des Systems 41
4.4 Modelchecking der Phasenautomaten mit MobyDC 42
4.4.1 Beweis der Korrektheit 43
4.4.2 Beweis der Vollständigkeit 44

5 Ausblick **47**

6 Literatur **48**

1 Einleitung

1.1 Ziel des Projektes

Innerhalb dieses individuellen Projektes soll ein Tool entwickelt werden, mit Hilfe dessen es möglich ist, Fehlerbäume in Phasenautomaten zu konvertieren.
Die Fehlerbäume sollen nach dem Standard [IEC93] in einer grafischen Benutzungsoberfläche gezeichnet werden können. Wenn ein Fehlerbaum dem Standard entsprechend gezeichnet worden ist, so soll es die Möglichkeit geben einen Export in einen Phasenautomaten durchzuführen. Das Format des genannten Exports soll von dem Modelchecker MobyDC gelesen werden können, um somit den Fehlerbaum, aus dem die Phasenautomaten entstanden sind auf Korrektheit und Vollständigkeit prüfen zu können.
Die Kernaufgabe des individuellen Projektes teilt sich somit mehrere Bestandteile. Die Erstellung eines Datenformates in der gewählten Programmiersprache Java zur Repräsentierung eines Fehlerbaums und eines Phasenautomaten sollte bearbeitet werden. Weiterhin ist die Entwicklung einer Arbeitsumgebung zum grafischen Entwickeln von standardgerechten Fehlerbäumen ein Teilgebiet der Arbeit sowie das Umsetzten der Abbildungsvorschrift von Fehlerbäumen auf Phasenautomaten.
Aus den genannten Bestandteilen der Kernaufgabe ergeben sich zwei Hauptanforderungen an das zu entwickelnde Tool:

1. Die erste ist das standardgerechte Zeichnen, Laden und Speichern von Fehlerbäumen.

2. Die zweite ist das Exportieren der Fehlerbäume in Phasenautomaten und Speichern dieser in einem von MobyDC, das von J.Tapken entwickelt wurde, lesbaren Dateiformat.

Weiterhin sollte das Tool ergonomisch zu bedienen sein sowie sich soweit wie möglich selbst erklären.

2 Grundlagen

2.1 Duration Calculus with Liveness(DCL)

Der Duration Calculus (DC) ist eine temporale Interval-Logik und wurde von C. Zhou, C. Hoare und A. Ravn in [CHR91] eingeführt.
Im DC können Aussagen darüber getroffen werden, wie lange ein Zustand in einem System angenommen wird.
Der Zustand des betrachteten Systems zu einem bestimmten Zeitpunkt wird im DC durch sogenannte Observablen, die ihren Wert im Laufe der zeit ändern, beschrieben.
Zustandszusicherungen, die mit π bezeichnet und durch die Grammatik

$$\pi ::= 0 \mid 1 \mid X = d \mid \neg\pi_1 \mid \pi_1 \wedge \pi_2.$$

erzeugt werden, treffen mit der Semantik

- $\mathcal{I}[\![0]\!](t) \stackrel{df}{=} 0$
- $\mathcal{I}[\![1]\!](t) \stackrel{df}{=} 1$
- $\mathcal{I}[\![X = d]\!](t) \stackrel{df}{=} \begin{cases} 1 & \text{falls } \mathcal{I}(X)(t) = d \\ 0 & \text{sonst} \end{cases}$
- $\mathcal{I}[\![\neg\pi]\!](t) \stackrel{df}{=} 1 - \mathcal{I}[\![\pi]\!](t)$
- $\mathcal{I}[\![\pi_1 \wedge \pi_2]\!](t) \stackrel{df}{=} \mathcal{I}[\![\pi_1]\!](t) \cdot \mathcal{I}[\![\pi_2]\!](t)$

eine Aussage über den Zustand eines Systems zu einem bestimmten Zeitpunkt.
DC-Terme, die von der Grammatik

$$\theta ::= x \mid \ell \mid \int \pi \mid f(\theta_1, \ldots, \theta_n)$$

erzeugt werden und mit θ abgekürzt werden, beschreiben mit der Semantik

- $\mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) \stackrel{df}{=} \mathcal{V}(x)$
- $\mathcal{I}[\![\ell]\!](\mathcal{V}, [b, e]) \stackrel{df}{=} e - b$
- $\mathcal{I}[\![\int \pi]\!](\mathcal{V}, [b, e]) \stackrel{df}{=} \int_b^e \mathcal{I}[\![\pi]\!](t)dt$
- $\mathcal{I}[\![f(\theta_1, \ldots \theta_n)]\!](\mathcal{V}, [b, e]) \stackrel{df}{=} \hat{f}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots (\mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e]))$

unter einer Interpretation $\mathcal{I}$ und einer Belegung $\mathcal{V}$ für ein Intervall $[b, e] \subseteq$ **time** die Zeit, die ein Zustand eine Zustandszusicherung erfüllt.
DC-Formeln werden von der Grammatik

$$D ::= \mathsf{true} \mid p(\theta_1, \ldots \theta_n) \mid \neg D \mid D_1 \wedge D_2 \mid \exists x : D \mid D_1 ; D_2$$

erzeugt und mit D bezeichnet. Ihre Semantik ist definiert durch

- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} \mathsf{true}$ immer
- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} p(\theta_1, \ldots, \theta_n)$ gdw. $\hat{p}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b,e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}[b,e])$
- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} \neg D_1$ gdw. $\mathcal{I}, \mathcal{V}, [b,e] \not\models_{DC} D_1$
- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} D_1 \wedge D_2$ gdw. $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} D_1$ und $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} D_2$
- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} \exists x : D$ gdw. Es existiert eine Belegung $\mathcal{V}'$ mit $\mathcal{V} =_{\backslash x} \mathcal{V}'$ und $\mathcal{I}, \mathcal{V}', [b,e] \models_{DC} D$
- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} D_1; D_2$ gdw. ein $m \in [b,e]$ existiert mit
  - $\mathcal{I}, \mathcal{V}, [b,m] \models_{DC} D_1$ und
  - $\mathcal{I}, \mathcal{V}, [m,e] \models_{DC} D_2$.

wobei $\mathcal{I}$ eine Interpretation, $\mathcal{V}$ eine Belegung und $[b,e] \subseteq$ **time** ein Intervall ist.
Der Duration Calculus with Liveness (DCL) aus [Ska94] erweitert den DC um zwei zusätzliche Chop-Operatoren. Somit ist die Semantik von DCL-Formlen, die mit A bezeichnet und von der Grammatik

$$A ::= \mathsf{true} \mid p(\theta_1, \ldots \theta_n) \mid \neg A \mid A_1 \wedge A_n \mid \exists x : A \mid A_1; A_2 | A_1 \lhd A_2 | A_1 \rhd A_2$$

erzeugt werden, gleich der von DC Formeln. Zusätzlich werden die beiden neuen Operatoren eingeführt.

- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} A_1 \lhd A_2$ gdw. ein $k \leq b$ existiert mit
  - $\mathcal{I}, \mathcal{V}, [k,b] \models_{DC} A_1$ und
  - $\mathcal{I}, \mathcal{V}, [k,e] \models_{DC} A_2$.
- $\mathcal{I}, \mathcal{V}, [b,e] \models_{DC} A_1 \rhd A_2$ gdw. ein $k \geq e$ existiert mit
  - $\mathcal{I}, \mathcal{V}, [b,k] \models_{DC} A_1$ und
  - $\mathcal{I}, \mathcal{V}, [e,k] \models_{DC} A_2$.

Mit diesen Operatoren werden in [Ska94] folgende Abkürzungen definiert:

- $\Diamond_L A \stackrel{df}{=} (\mathsf{true}; A; \mathsf{true}) \rhd \mathsf{true}$ (echtes eventually)
- $\Box_L A \stackrel{df}{=} \neg \Diamond_L \neg A$ (echtes always)

Wir verwenden den DCL, da wir für die formale Semantik von Fehlerbäumen ein echtes eventually benötigen.

2.2 Fehlerbaumanalyse

2.2.1 Intention

Fehlerbäume werden in den Ingenieurswissenschaften eingesetzt um sicherheitskritische Systeme zu analysieren. Hauptanwendungsgebiet ist die Atom- und Luftfahrtindustrie. Innerhalb eines Fehlerbaums wird ein zu vermeidendes Hauptereignis dargestellt, z.B. der Absturz eines Flugzeugs oder das Explodieren eines Tanks.
Dieses Hauptereignis wird innerhalb eines Fehlerbaums schrittweise in seine Ursachen zerlegt. Somit führt man das Grundereignis, das man untersucht, auf seine Ursachen zurück. Fehlerbäume besitzen eine formale Semantik im DCL.
In [RST00] und [RST01] werden für die Analyse der Semantik eines Fehlerbaumes zwei Aspekte betrachtet. Diese Aspekte sind die Korrektheit und Vollständigkeit der Zerlegung des Hauptereignisses in seine Ursachen. Wenn wir später einen Fehlerbaum auf Korrektheit und auf Vollständigkeit überprüfen wollen, so wird die Korrektheit und die Vollständigkeit jedes einzelnen Gatters mit Hilfe der aus diesem Gatter enstandenen Phaseautomaten in MobyDC geprüft.

2.2.2 Gattertypen und Events

Verknüpfungen innerhalb eines Fehlerbaums stellt man über gerichtete Kanten her.
Wenn ein Ereignis in zwei Ursachen Ereignisse zerlegt wird, so muss festgelegt werden, wie die beiden Ursachen mit einander in Beziehung stehen. Diese Beziehung definiert man über sogenannte Gatter. Durch den Typ des Gatters wird spezifiziert, wie die Ursachen eintreten müssen, damit auch das übergeordnete Ereignis eintritt.
Man benutzt 5 Gattertypen:

1. **AND - Gatter** Das *AND* Gatter ist vergleichbar mit einem einfachen und. Das bedeutet, wenn das übergeordnete Ereignis eintreten soll, so müssen alle mit dem *AND* Gatter verbundenen Ereignisse eintreten, es besteht aber keine zeitliche Beziehung zwischen dem Eintreten der Ereignisse.

2. **OR - Gatter** Das *OR* Gatter verhält sich ähnlich wie das *AND* Gatter jedoch muss hier nur eines der untergeordneten Ereigsnisse eintreten, damit das übergeorndete eintritt. Es existiert ebenfalls kein zeitlicher Zusammenhang zwischen den Ereignissen.

3. **AND-CC - Gatter** Beim *AND-CC* Gatter besteht eine zeitliche Beziehung zwischen den Ereignissen. Diese besagt, dass das übergeordnete Ereignis irgendwann eintreten muss, nachdem alle untergeordneten Ereignisse irgendwann eingetreten sind.
   Dies kann bedeuten, dass zu dem Zeitpunkt, zu dem das übergeordnete Ereignis eintritt, eines der untergeordneten Ereignisse schon nicht mehr gilt, aber es zuvor gegolten hat und deswegen wird das übergeordnete oder auch Top-Level Ereignis ausgelöst.

4. **AND-CC-sync - Gatter** Bei einem *AND-CC-sync* gibt es ebenfalls eine zeitliche Beziehung zwischen den Ereignissen die eintreten müssen um ein weiteres

auszulösen. Das *AND-CC-sync* Gatter fordert jedoch, dass diese Ereignisse parallel eintreten und nicht irgendwann. Das Top-Level Ereignis des *AND-CC-sync* Gatters tritt ein, nachdem alle Unterereignisse parallel eingetreten sind.

5. **OR-CC - Gatter** Das *OR-CC* Gatter stellt auch eine zeitliche Relation zwischen den Ereignissen her. Hier wird ausgesagt, dass das übergeordnete Ereignis direkt eintritt, nachdem eine der Ursachen eingetreten ist.

Wobei hier CC für *Cause-Consequence* also Ursache-Wirkungs-Beziehung steht.
In [RST02] wird eine andere Namensgebung für die Gatter eingeführt. Dort existieren die Gatter *Dand, Dor, Cand, ACand* und *Cor* wobei dort das *D* als Präfix benutzt wird, wenn keine Ursache-Wirkungs-Beziehung besteht und somit nur eine Dekomposition vorliegt. Das *C* steht hier ebenfalls für Cause Consequence also Ursache Wirkung und das *A* in *ACand* steht für die Asynchronität der Ursachen.
Somit sind die Gatter aus [RST02] folgendermaßen zu verstehen:
Dor = *AND*
Dand = *DOR*
Cand = *AND-CC-sync*
ACand = *AND-CC*
Cor = *OR-CC*
Da jedem Event, im Gegensatz zu den anderen Gattern, eine DCL-Formel zugeordnet wird, unterscheidet sich in diesem Projekt die Darstellung von Event-Gattern von der Darstellung der Anderen. Jedem Event wird ein Name und eine DCL-Formel gegeben. Die grafische Darstellung eines Events ist in Abbildung 1 zu sehen.
In [IEC93] wird ein internationaler Standard zur Form von Fehlerbäumen gegeben. Die

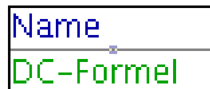

Abbildung 1: Darstellung eines Event-Gatters

Verwendung von Symbolen bleibt nach diesem Standard jedoch relativ offen, da auch eigene Symbole verwendet werden dürfen.
Innerhalb dieses Individuellen Projektes finden die Symbole in Abbildung 2 für die entsprechenden Gatter Verwendung

2.2.3 Beispiel

Das einfachste Beispiel eines Fehlerbaums ist, wenn ein unerwünschtes Ereignis direkt auf nur ein anderes Ereignis zurückzuführen ist. Dies kann z.B. der Fall sein, wenn eine Pumpe dadurch zerstört wird, dass ein Ventil zu lange geschlossen ist, wie in Abbildung 3 dargestellt.
Dieser Fehlerbaum enthält nur Events und durch die vorhandene Kante wird dargestellt, wie diese beiden Events in Beziehung stehen. In dem in Abbildung 3 dargestellten

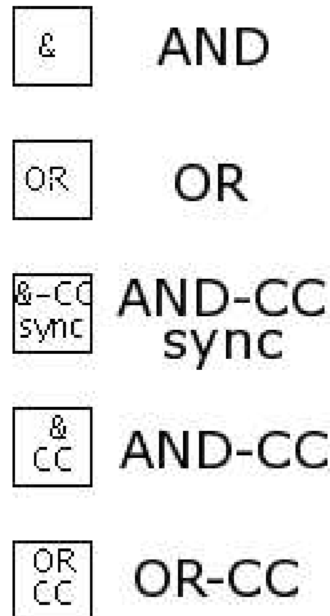

Abbildung 2: Gatter eines Fehlerbaums

Fehlerbaum bedeutet die Kante, dass die Pumpe explodiert, wenn das Ventil länger als 10 Zeiteinheiten geöffnet ist.

2.2.4 DCL-Syntax in Faulttree2Phaseautomata

Die Syntax der DCL-Formeln übersteigt die Darstellungsmöglichkeiten einer normalen Rechner-Tastatur. Da z.B. der Operator $\Diamond$ nicht mit Standard-Zeichen darstellbar ist, aber vor jeder DCL-Formel, die hier Verwendung findet, vorkommt, setzten wir diesen als gegeben voraus und verlangen vom Benutzer nur die Eingabe des in jedem Event geltenden DCL-Ausdruckes π in eckigen Klammern. Optional kann dann noch ein Zeitausdruck mit >= , <=, > oder < hinzugefügt werden.
Eine DCL-Formel $\Diamond(\lceil\pi\rceil \wedge b \leq \ell)$ hätte dann in der vereinfachten Syntax die Form $[\pi] >= b$.

2.3 Phasenautomaten

Phasenautomaten sind Automaten die eine formale Semantik im DC besitzen. Die Definition wird in [Tap01] gegeben. Hier werden Phasenautomaten als Zwischensprache von DC Spezifikationen eingeführt.
Ein Phasenautomat besteht aus verschiedenen Zuständen, den sogenannten Phasen. Jeder Phase wird ein DC Ausdruck zugeordnet, was bedeutet, dass wenn sich der Automat in einer Phase befindet, zu diesem Zeitpunkt der Zustandsausdruck der Phase gilt.

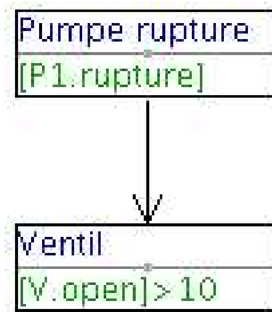

Abbildung 3: einfacher Fehlerbaum

2.3.1 Definition der Syntax

Die formale Definition von Phasenautomaten in [Tap01] sagt, dass ein Phasenautomat $\mathcal{A}$ ein Tupel $\mathcal{A} = (P, E, C, cl, s, d, P_0)$ ist, wobei

- P eine endliche Menge von Zuständen,
- $E \subseteq P \times P$ die Transitionsrelation,
- C eine endliche Menge von Uhren,
- $cl : P \rightarrow 2^C$ eine Abbildung, die jedem Zustand eine Menge von Uhren zuordnet,
- $s : P \rightarrow SA(Obs)$ eine Abbildung, die jedem Zustand einen Zustandsausdruck zuordnet,
- $d : C \rightarrow IE$ eine Abbildung, die jeder Uhr einen Intervallausdruck zuordnet, der auch Variablen enthalten kann und die Verweildauer in dem entsprechenden Uhrenbereich beschränkt und
- $P_0 \subseteq P$ eine Menge von Startzuständen ist.

2.3.2 Uhren in Phasenautomaten

In Phasenautomaten existieren neben Kanten und Phasen Uhren. Der Wert dieser Uhren muss jeweils innerhalb eines Intervalls von zwei Zeiteinheiten liegen, damit der zu der Uhr gehörende Zustand gültig ist. Das mit einer Uhr beschriebene Intervall beschränkt somit die Zeit, die eine Phase in einem Automat gelten kann.
In [Tap01] wird beschrieben, dass einer Phase mehrere Uhren zugeordnet werden können. Das Programm MobyDC geht bei der Modellierung von Phasenautomaten jedoch davon aus, dass jedem Zustand genau eine Uhr zugeordnet wird.
In [Tap01] wird jedoch gezeigt, dass jeder Phasenautomat in einen äquivalenten Automaten mit genau einer Uhr für jede Phase umgewandelt werden kann.

2.3.3 Beispiel

Ein einfaches Beispiel für einen Phasenautomaten ist der Automat zu dem DCL Ausdruck „irgendwann gilt nur noch a“aus [Sch02], der in Abbildung 4 gezeigt wird. Diese Abbildung wurde aus MobyDC exportiert. Daher enthält sie neben den beiden Phasen $P0$ und $P1$ auch noch einen sogenannten Start Port. Dieser Port ist keine Phase. Alle Phasen, die mit diesem Start Port mit Kanten verbunden sind, sind Startzustände.

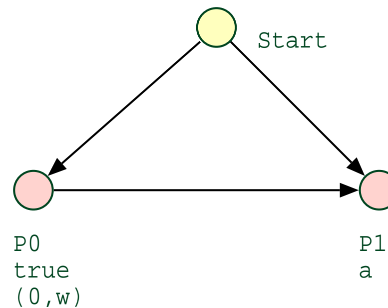

Abbildung 4: einfacher Phasenautomat zu „irgendwann gilt a“

2.4 MobyDC

Das Programm MobyDC wurde von J.Tapken [Tap01] entwickelt und bietet die Möglichkeit hierachische Phasenautomaten zu zeichnen und diese zu Netzwerken zu verbinden. Diese Netzwerke werden kurz Nopas genannt (Network of Phase Automata). Weiterhin bietet MobyDC die Möglichkeit diese Nopas mit dem Model-Checker MC4DC darauf zu testen, ob sie einen gemeinsamen Ablauf haben. Man sagt auch, dass die Nopas auf Leerheit getestet werden.

2.5 Model-Checking in MobyDC

Die Möglichkeit des Model-Checking ist ein zentrales Ziel dieser Arbeit und wird später an Hand von Beispielen demonstriert.
Bei der Verifikation von Fehlerbäumen mit DC-Semantik wird die Vollständigkeit und die Korrektheit des zu untersuchenden Fehlerbaums überprüft. In [Sch02] wird geschildert welche Phasenautomaten man auf Leerheit testen muß, um die untersuchte Eigenschaft zu verifizieren.
Generell benötigt man zu jedem Event Gatter einen Phasenautomaten und sein Komplement. Weiterhin müssen zusätzliche Automaten zu den Gattern, die die Events verknüpfen erstellt werden. Diese Automaten sollen von *Faulttree2Phaseautomata* konstruiert werden. Außer diesen Automaten wird für die Verifikation noch ein System Modell benötigt, welches das Verhalten der Systemkomponenten mit Phasenautomaten modelliert. Diese Automaten müssen vom Benutzer selbst erstellt werden.
Nach [Gór94] und [Bec83] kommen in Fehlerbäumen nur drei Arten von Events vor,

die durch drei verschiedene Arten von DCL-Formeln beschrieben werden können. Diese Ereignis-Muster werden als Patterns bezeichnet und lassen sich wie folgt beschreiben:

- Das System nimmt einen bestimmten Zustand an.
- Das System nimmt einen Zustand an, und dieser Zustand wird nicht mehr verlassen, wenn er einmal angenommen wurde.
- Eine Sequenz von Zuständen wird durchlaufen.

Diese Pattern lassen sich durch folgende DCL-Formeln beschreiben:

- $\Diamond(\lceil \pi \rceil \wedge b \leq \ell)$ (Erreichbarer Zustand)
- $\Diamond \Box \lceil \pi \rceil \wedge \neg \Diamond(\lceil \pi \rceil ; \lceil \neg \pi \rceil)$ (Endzustand)
- $\Diamond(\lceil \pi_1 \rceil \wedge b_1 \leq \ell \leq e_1; \ldots ; \lceil \pi_n \rceil \wedge b_n \leq \ell \leq e_n)$ oder $\lceil \pi_1 \rceil \wedge b_1 \leq \ell \leq e_1; \ldots ; \lceil \pi_n \rceil \wedge b_n \leq \ell \leq e_n$ (Sequenz)

Um eine Überprüfung der Korrektheit und Vollständigkeit von Fehlerbäumen durchzuführen benötigt man komplementierbare Phasenautomaten. In [Sch02] wird gezeigt, dass eine Teilklasse der Phasenautomaten existiert, die gegenüber dem Komplement abgeschlossen ist. Dies sind die deterministischen Phasenautomaten.
Die Phasenautomaten zu diesen Events sind keine deterministischen Automaten und daher nicht nach der in [Sch02] vorgstellten Konstruktion komplementierbar. Daher werden explizit die Komplemente der Phasenautomaten zu diesen Patterns angegeben. Weiterhin wird dort beschrieben, welche Automaten benötigt werden, um für jedes der verwendbaren Gatter Vollständigkeit und Korrektheit zu zeigen. Diese Konstruktionsvorschriften bilden die Grundlage für die Exportfunktion in *Faulttree2Phaseautomata*.

2.5.1 Pattern 1 - erreichbarer Zustand

Die Abbildung 5 zeigt den Phasenautomaten $\mathcal{A}_Z$ zu der DCL-Formel $\Diamond \lceil \pi \rceil \wedge b \leq \ell$. In

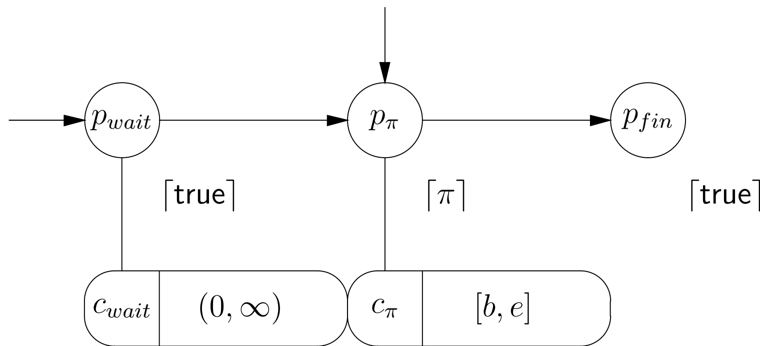

Abbildung 5: Automat $\mathcal{A}_Z$ zum 1. Pattern (erreichbarer Zustand)

Abbildung 6 wird das entsprechende Komplement zu Pattern 1 gezeigt. Dieser Automat $\overline{\mathcal{A}_Z}$ entspricht $\neg[\Diamond \lceil \pi \rceil \wedge b \leq \ell \wedge \ell \leq e]$.

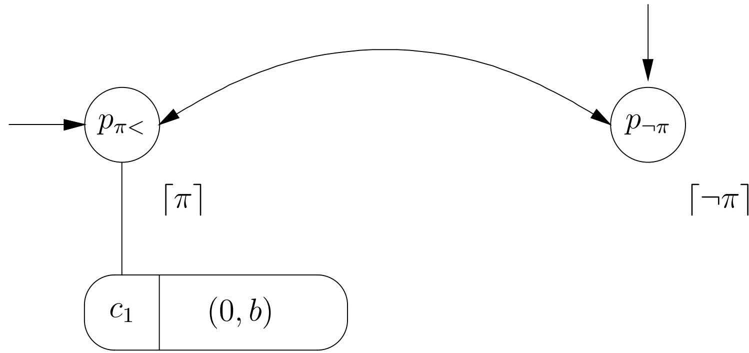

Abbildung 6: Komplementautomat $\overline{\mathcal{A}_Z}$ zum 1. Pattern (erreichbarer Zustand)

2.5.2 Pattern 2 - Endzustand

Der Automat $\mathcal{A}_E$ zum Endzustands-Pattern modelliert die DCL-Formel $\Diamond\Box\lceil\pi\rceil \wedge \neg\Diamond(\lceil\pi\rceil;\lceil\neg\pi\rceil)$ und wird in Abbildung 7 gezeigt. Der Komplementautomat $\overline{\mathcal{A}_E}$ ist in

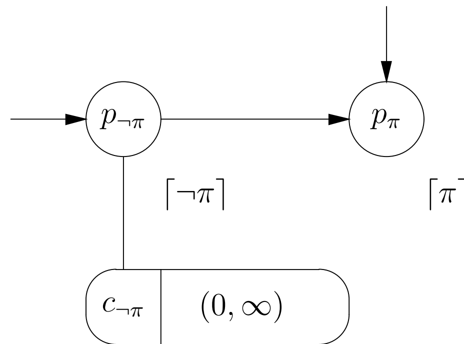

Abbildung 7: Automat $\mathcal{A}_E$ zum 2. Pattern (Endzustand)

Abbildung 8 zu sehen.

2.5.3 Pattern 3 - Sequenz

Der Automat $\mathcal{A}_S$ zu $\Diamond(\lceil\pi_1\rceil \wedge b_1 \leq \ell \leq e_1; \ldots; \lceil\pi_n\rceil \wedge b_n \leq \ell \leq e_n)$ hat die in Abbildung 9 dargestellte Form. Da der Automat zu Pattern 3 ein nicht komplementierbarer Automat ist, existiert nicht zu jedem Automaten zum Sequenz-Pattern ein Komplement. Aus diesem Grund wird einschränkend vorausgesetzt, dass jeweils zwei Zustandsausdrücke einander ausschließen, d.h.

$$\pi_i \wedge \pi_j \equiv \mathsf{false} \quad \text{für alle } 1 \leq i \neq j \leq n$$

Die Konstruktionsidee für die so eingeschränkte Klasse der Automaten zum Sequenz Pattern wird in [Sch02] vorgestellt. Dort werden ebfalls die Beweise für die Korrektheit der Konstruktionen zu den einzelnen Pattern gegeben, was Voraussetzung dafür ist, dass sie in diesem Projekt verwendet werden.

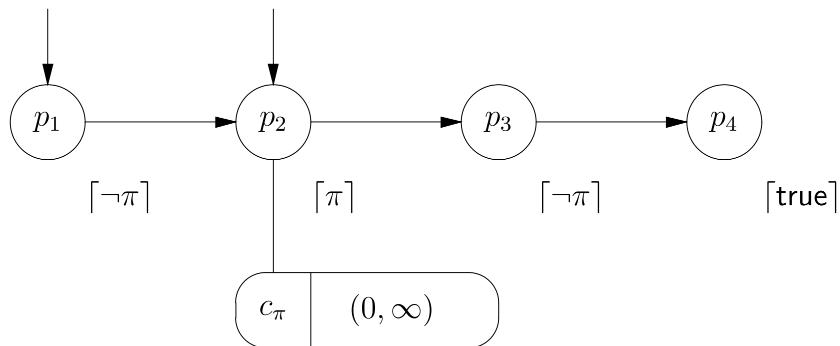

Abbildung 8: Komplementautomat $\overline{\mathcal{A}_E}$ zum 2. Pattern (Endzustand)

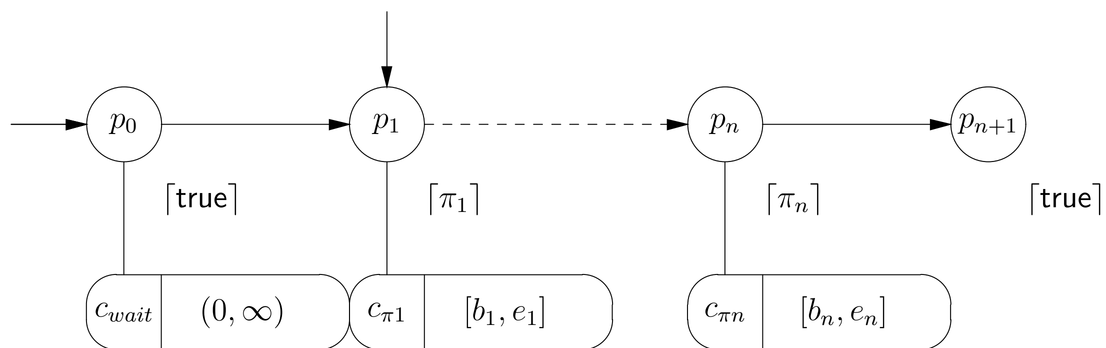

Abbildung 9: Automat $\mathcal{A}_S$ zum 3. Pattern (Sequenz)

3 Umsetzung

3.1 Vorüberlegung

3.1.1 Vorgehensweise

Die Vorüberlegung zur Erstellung des geforderten Java-Tools ergaben, dass es sehr sinnvoll sei, die herausgearbeiteten Anforderungen an das individuelle Projekt in zwei Schritten umzusetzten.
Zunächst wurde ein Prototyp entwickelt, der es möglich machte, Fehlerbäume zu zeichnen, zu speichern und zu laden.
Danach wurde dieser Prototyp um die Exportfunktion in Phasenautomaten in einem XML-Format und einem von MobyDC lesbarem Format erweitert.
Durch das beschriebene Vorgehen wurden die Anforderungen an das individuelle Projekt nach einander erfüllt.

3.1.2 Namensgebung

Der Name des zu entwickelnden Java Tools *Faulttree2Phaseautomata* entstand aus der Überlegung heraus, dass der Nutzen des Tools darin besteht, Fehlerbäume in Phasenautomaten umzuwandeln. Jedoch kann man ohne MobyDC mit den epxortierten Phasenautomaten nicht viel anfangen. Somit ist das Tool völlig alleine für den Endanwender nicht sehr nützlich und ist daher auch nur ein sogenanntes Tool und kein selbstständiges Programm. Bei Tools die eine Umwandlung von einem Datenformat in ein anderes Umwandeln ist es üblich den Namen aus den Namen der Datenformate verbunden mit dem englischen Wort „to "zu synthetisieren. Somit wäre der Name des in diesem individuellen Projektes zu erstellenden Tools „Faulttree to Phaseautomata "weil es Faulttrees in Phaseautomata umwandelt. Die 2 im Namen kommt dadurch zustande, dass man zwei im englischen two schreibt und two bei der Aussprache einen sehr ähnlichen Klang wie to hat. Wenn man also „Faulttree to Phaseautomata"englisch ausspricht und „Faulttree two Phaseautomata"ebenfalls, so klingen die beiden Ausdrücke identisch. Anstatt two schreibt man nun noch 2 und erhält so *Faulttree2Phaseautomata*.

3.2 Verwendete Techniken

3.2.1 Java

Java [Sun] wurde als Programmiersprache gewählt, weil Java durch die Plattformunabhängigkeit optimale Voraussetzungen für den praktischen Einsatz von *Faulttree2Phaseautomata* bietet. So ist es durch Java möglich *Faulttree2Phaseautomata* auf jedem System auszuführen, das Java unterstützt.

3.2.2 JGraph

Da *Faulttree2Phaseautomata* in Java geschrieben werden sollte und außerdem gefordert war, dass Fehlerbäume gezeichnet werden können, war es sinnvoll auf eine Java-Bibliothek zurück zu greifen, die das Zeichnen von Graphen unterstützt. Es gibt mehrere Anbieter solcher Bibliotheken. Nach einer kurzen Sichtung der Vor- und Nachteile einzelner Anbieter fiel die Wahl auf das Open-Source Projekt JGraph[Jgr], da diese Bibliothek leicht erweiter- und vor allem an die speziellen Anforderungen von Fehlerbäumen anpassbar ist.

3.2.3 JFlex

In Fehlerbäumen werden Event-Gattern ein Name und eine DCL-Formel zugeordnet. Dies ist natürlich auch in *Faulttree2Phaseautomata* möglich, wie Abbildung 1 zeigt. Die DCL-Formeln der Event Gatter werden beim Export in Phasenautomaten analysiert und beeinflussen die Gestalt des Phasenautomaten, der erzeugt wird. Zur Analyse dieser vom Benutzer eingegebenen DCL-Formeln wurde ein sogenannter Lexer benötigt. Dieser Lexer soll dazu dienen, die Syntax der Formeln auf Korrektheit zu überprüfen. Weiterhin müssen die Formeln, wenn sie syntakisch korrekt sind, semantisch untersucht werden, wozu der Lexer die einzelnen Bestandteile einer DCL-Formel extrahiert und als einzelne Token zurückliefert.
Der Lexer JLex [JLe] bot sich zunächst hierfür an, da er direkte Java Kommandos unterstützt. Um diesen Lexer zu benutzen, muss man in einer eigenen Syntax mit Hilfe von regulären Ausdrücken die einzelnen Token, die vom Lexer identifiziert werden sollen, spezifizieren. Danach wird dann aus dieser Spezifikation eine Java Datei erzeugt, die man im Sourcecode seines Programmes verwerten kann. JLex ist ein sehr mächtiger Lexer, mit dem theoretisch die lexikographische Analyse eines komplexen Compilers, wie dem des Java Compilers selber spezifiziert werden könnte.
Dieser Umfang und die etwas komplexere Einrichtung dieses Lexers ließ die Entscheidung auf den ebenfalls frei erhältlichen Lexer JFlex [JFl] fallen.
JFlex enthält die wesentlichen Elemente die auch JLex enthält, ist aber leichter einzurichten, komfortabler zu bedienen und laut Hersteller dazu gedacht schnell und einfach einen Lexer zu erstellen, was genau auf die Anforderungen an den gesuchten Lexer passte.

3.3 Oberflächendesign

Das Tool *Faulttree2Phaseautomata* wurde als Java Applikation mit einer grafischen Benutzeroberfläche zur Bedienung entwickelt. Die Elemente, die in diese Benutzeroberfläche Verwendung fanden, sind Standardelemente aus dem Java Paket *javax.swing*.
Die Bedienelemente des Editors für Fehlerbäume sind in zwei Bereiche aufgeteilt. Der erste Bereich fügt dem Fehlerbaum ein Gatter oder ein Event hinzu. Der zweite Bereich beinhaltet Funktionen wie Speichern, Laden und Exportieren. Da der zweite Bereich teilweise Standard-Funktionen enthält, wurde als Basis für den *Faulttree2Phaseautomata* Editor ein Editor aus einem Beispiel von [Jgr] verwendet und angepasst. Die Abbildung

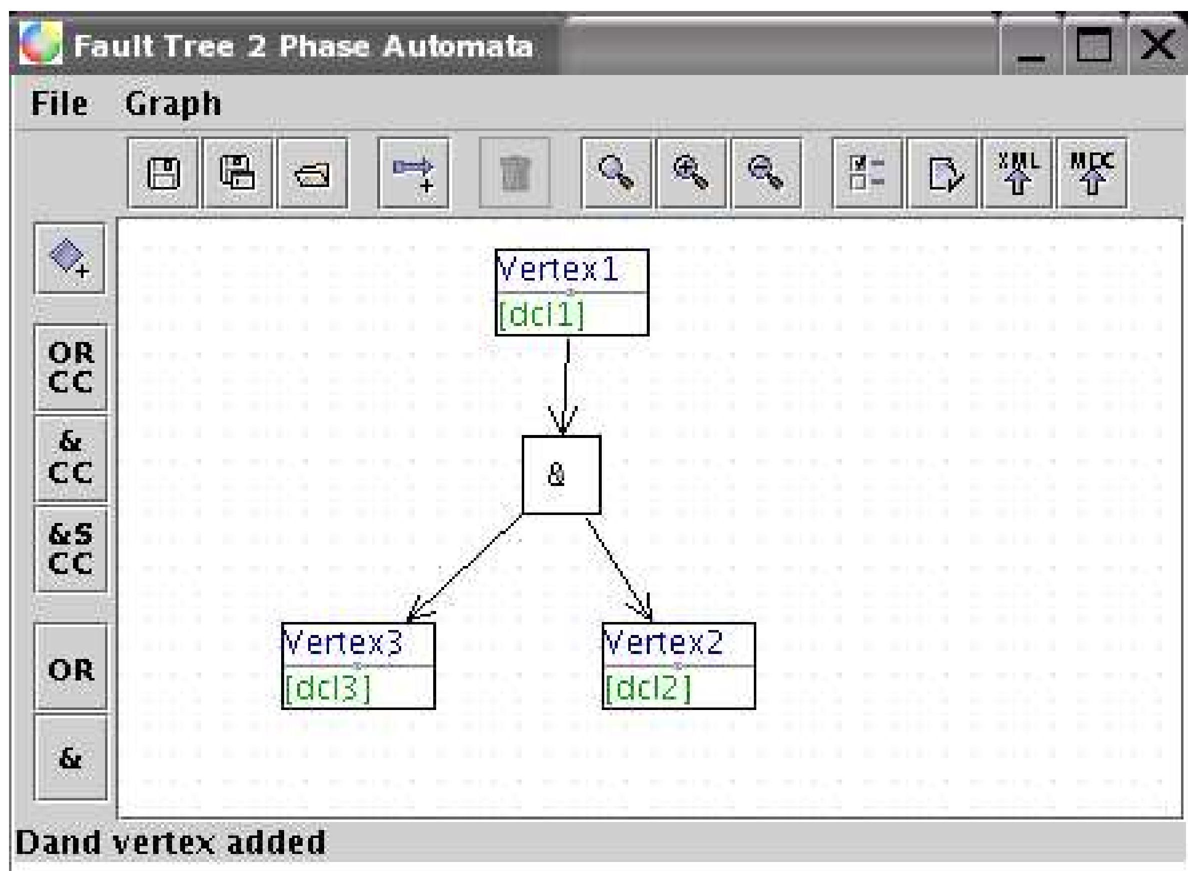

Abbildung 10: Hauptfenster des Tools

10 zeigt das Hauptfenster von *Faulttree2Phaseautomata*. In der Mitte ist die weiße Zeichenfläche zu sehen. Am linken Rand befindet sich eine Toolbar, die als Buttons die Funktionen enthält, die dem Fehlerbaum ein Gatter oder ein Event hinzufügen. Am oberen Rand befindet sich ebenfalls als Toolbar der Rest der Funktionen.

3.3.1 Textmenü

Weiterhin enthält *Faulttree2Phaseautomata* ein Textmenü, welches die Kategorien File und Graph enthält. Dieses Menü ist in Abbildung 10 am oberen Rand des Hauptfensters zu sehen. Außerdem ist auf der Abbildung noch die Statusleiste zu sehen, in welcher Aktuelle Meldungen über das Bearbeiten des aktuellen Fehlerbaums ausgegeben werden.
Die Zeichenfläche ist mit einem sogenannten Grid ausgestattet. Dieser Grid ist optisch an kleinen Markierungen in der weißen Zeichenfläche zu erkennen. Der Grid ist sozusagen das kleinste Maß für die Entfernung zweier Objekte die nicht null ist. So können Gatter auf der Zeichenfläche nicht völlig frei sondern immer nur entlang dieses Grids verschoben werden. Dies sorgt für eine gewisse Ordnung und Übersicht innerhalb der Zeichnung. Es ist so z.B. möglich zwei Objekte frei Hand ohne Maßangaben exakt entlang einer Linie zu positionieren.
Die Kategorie File im Textmenü enthält wie auf Abbildung 11 zu sehen folgende Un-

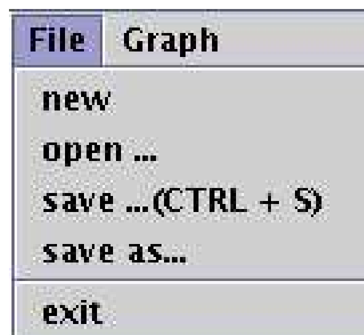

Abbildung 11: Textmenü Kategorie File

terpunkte:

- *new* löscht alle sichtbaren Objekte, die sich im aktuellen Fehlerbaum befinden.
- *open* zeigt einen Dialog zum Öffnen eines gespeicherten Fehlerbaums.
- *save* speichert den aktuellen Fehlerbaum. Wenn der Fehlerbaum bereits einmal gespeichert worden ist oder aus einer Datei geöffnet wurde, wird er erneut in diese Datei geschrieben. Wenn der Fehlerbaum noch nicht gespeichert wurde und nicht geöffnet worden ist, dann wird ein Dialog angezeigt um eine Datei auszuwählen in die der Fehlerbaum geschrieben werden soll.
- *save as* öffnet sofort den Dialog zum Auswählen einer Datei, in die der aktuelle Fehlerbaum geschrieben werden soll.

- *exit* schließt das Tool sofort ohne den aktuellen Fehlerbaum zu speichern.

Die Kategorie Graph im Textmenü enthält die in Abbildung 12 dargestellten Menüpunkte. Hinter diesen Menüpunkten verbergen sich die beiden Hauptfunktionen von *Fault-*

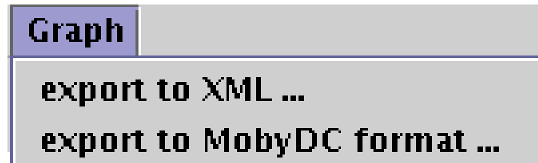

Abbildung 12: Textmenü Kategorie Graph

tree2Phaseautomata, denn hiermit werden die Dialoge zum exportieren von Fehlerbäumen nach Phasenautomaten geöffnet. Diese Dialoge sind sich sehr ähnlich. In Abbildung

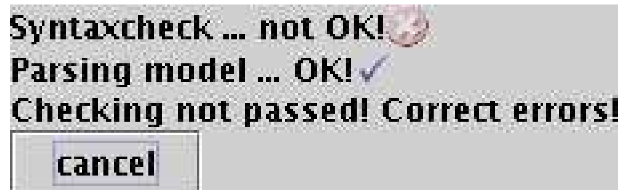

Abbildung 13: Export Graph Dialog

13 wird einer dieser Dialoge gezeigt. Der Dialog zeigt, dass jeweils die Synatx der DCL Formeln der Events überprüft wird und ebenfalls versucht wird den gezeichneten Fehlerbaum auf das hinterlegte Model für Fehlerbäume zu parsen. Nur wenn beides gelingt und keine Fehler auftreten wird in diesem Dialog die Möglichkeit geboten den Export über einen zusätzlichen export Button zu starten. In dem in Abbildung 13 gezeigten Dialog steht dieser Export Button nicht zur Verfügung, da die Überprüfung der Syntax Fehler ergeben hat.

3.3.2 Obere Toolbar

Abbildung 14: obere Toolbar

Die obere Toolbar, die in Abbildung 14 gezeigt wird, bietet dem Benutzer die Möglichkeit den gezeichneten Fehlerbaum zu bearbeiten und andere Funktionen auszuführen. Hier findet der Benutzer Buttons zu den Funktionen *save*, *save as*, *open*, *export to XML* und *export to MobyDC format* die ebenfalls im Textmenü vorhanden sind und bereits

beschrieben wurden. Die Buttons in der Toolbar sollen dem Benutzer den Aufruf der entsprechenden Funktionen vereinfachen und die Arbeit beschleunigen.
Weiterhin enthält die Toolbar Buttons zum An- und Ausschalten des Verbindungsmodus, Vergrößern und Verkleinern der Ansicht und manuellen Auslösen des Syntax Checking und des Parsens auf das Fehlerbaummodel.
Der Verbindungsmodus dient dazu dem Fehlerbaum Kanten hinzuzufügen. Ist der Verbindungsmodus aktiviert, so ist bei jedem Gatter und jedem Event ein sogenannter Port sichtbar. Bewegt der Benutzer die Maus über diesen Port, so erscheint ein anderer Mauszeiger und über das Drag and Drop Verfahren kann der Benutzer nun Kanten von einem Port zu einem anderen Port ziehen und somit zwei Gatter oder Events mit einander verbinden, wie in Abbildung 15 zu sehen. Die Buttons zum Vergrößern und Verkleinern

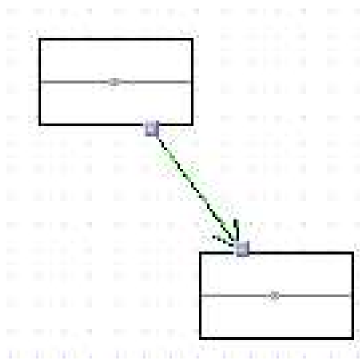

Abbildung 15: Verbindung zweier Events

der Ansicht dienen dazu die Übersicht für den Benutzer zu erhöhen. So ist es möglich große Fehlerbäume zu überblicken indem man die Ansicht verkleinert und somit mehr Objekte auf gleicher Fläche darstellen kann.
Das manuelle Auslösen des Syntax Checkers soll dem Benutzer die Möglichkeit einräumen während der Entwicklung eines Fehlerbaums die Syntax der bisher eingefügten DCL-Formeln zu überprfen. Als Ergebnis dieser Funktion wird dem Benutzer ein Dialog angezeigt, der in Abbildung 16 gezeigt wird. Dieser Dialog zeigt an, ob ein Fehler in der

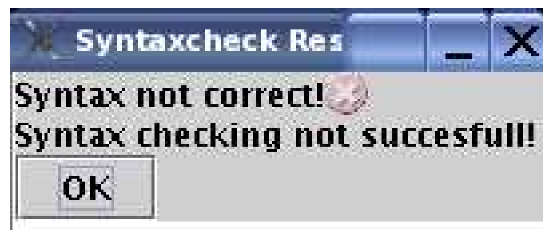

Abbildung 16: Syntax Checker Dialog

Syntax einer Formel eines Events vorliegt.
Das manuelle Auslösen des Modelparsers dient dazu dem Benutzer während der Entwicklung die Möglichkeit zu geben, zu überprüfen, ob der bis jetzt gezeichnete Fehlerbaum

Fehler in seinem Aufbau enthält, die dazu führen könnten, dass ein korrekter Export in Phasenautomaten nicht möglich wäre. Der Ergebnisdialog wird in Abbildung 17 abgebildet. In diesem Dialog erhält der Benutzer einen detailierten Überblick über die

Abbildung 17: Modelparser Dialog

Ergebnisse des Parsens des Fehlerbaums.
Zuerst wird überprüft ob der Fehlerbaum zyklenfrei und zusammenhängend ist. Weiterhin wird getestet ob das Top-Level Event wirklich ein Event ist und kein anderes Gatter, denn das Top-Level Event ist das Ereignis, welches innerhalb des Fehlerbaums untersucht werden soll. Als letztes wird überprüft, ob es innerhalb des Fehlerbaums ein Event gibt, dass nicht das Top-Level Event des Fehlerbaums ist und dem eine DCL-Formel zugeordnet wurde, die dem zweiten, dem Endzustands Pattern, entspricht. Einem anderen Event als dem Top-Level Event eine DCL-Formel des Endzustands Typs zu zuordnen wäre nicht sinnvoll, da das diesem Event übergeordnete Event niemals ausgelöst werden kann, da das System niemals den Zustand des Events verlässt, das dem Endzustands Pattern entspricht.

3.3.3 Linke Toolbar

Die linke Toolbar, die in Abbildung 18 gezeigt wird beinhaltet Buttons die dem aktuellen Fehlerbaum Events und Gatter hinzufügen. Die Gatter und Events tauchen nach betätigen des entsprechenden Buttons in der linken oberen Ecke der Zeichenfläche auf.

3.3.4 Editieren eines Events

Da den Events Eigenschaften, wie ein Name und eine DCL-Formel zugeordnet werden gibt es in *Faulttree2Phaseautomata* einen Dialog, um diese Eigenschaften zu erfassen. Dieser Dialog wird in Abbildung 19 dargestellt. Er zeigt Eingabefelder für den Namen und die DCL-Formel, sowie eine Auswahlbox für den Typ der DCL-Formel.

3.4 Programm-Architektur

Die Architektur von *Faulttree2Phaseautomata* entspricht im wesentlichen einer typischen 3-Schichten-Architektur, in der es eine Schicht für die Benutzeroberfläche, eine für die

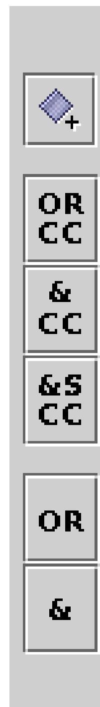

Abbildung 18: linke Toolbar

Abbildung 19: Dialog zum Erfassen der Eigenschaften eines Events

Logik des Programms und eine für die Datenhaltung gibt. Der Vorteil einer solchen Architektur ist, dass es durch diesen modularen Aufbau des Programms theoretisch möglich ist die einzelnen Schichten auszutauschen. Das bedeutet, falls z.B. eine andere Art von Datenhaltung gewünscht wird, kann man die Schichten der Benutzeroberfläche und der Logik beibehalten und implementiert nur eine neue Datenhaltungsschicht.
Diese 3 Schichten Architektur spiegelt sich auch im Sourcecode von *Faulttree2Phaseautomata* wieder. Es existiert ein Java Paket zur Benutzeroberfläche und eines zur Logik des Programms. Da die Datenhaltung in *Faulttree2Phaseautomata* durch die Verwendung einer Standard-Bibliothek auf ein Minimum an Sourcecode reduziert werden konnte, wurde die Datenhaltung als Unterpaket in die Logikschicht integriert, wie in Abbildung 20 zu sehen ist. Dieses Paket Diagramm stellt alle Java-Pakete von *Faulttree2Phaseautomata* dar und gibt somit einen guten Überblick über die Struktur des Sourcecodes.

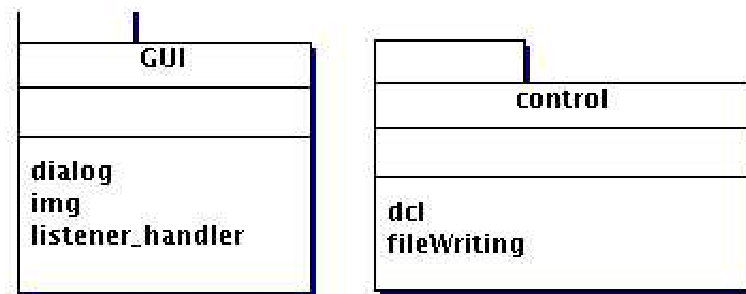

Abbildung 20: Java Pakete von F2P

3.4.1 Entwurf der Oberfläche

Die in diesem Projekt verwendete Bibliothek JGraph [Jgr] beeinhaltet zwei wesentliche Komponenten. Zum Ersten die grafische Komponente, die das Anzeigen von Graphen ermöglicht und einfach wie eine Standard-Grafik-Komponente in die Oberfläche eingebunden werden kann. Zum Zweiten eine Logik Komponente, die das Datenmodell des Graphen, der dargestellt wird, repräsentiert.
Das Java-Paket in *Faulttree2Phaseautomata*, dass die Oberflächenschicht repräsentiert ist mit *GUI* (engl: graphical user interface) abgekürzt. Innerhalb dieses Paketes existiert eine Klasse, die *MainFrame* heißt und die das Hauptanwendungsfenster des Tools erzeugt. Innerhalb dieser Klasse werden im Konstruktor die beiden Toolbars, das Textmenü und die Zeichenfläche initialisiert. Die Zeichenfläche ist die GUI-Komponente von JGraph. Weiterhin werden andere Parameter wie z.B. der Titel des Frames gesetzt.
Abbildung 21 zeigt einen Ausschnitt des Klassendiagramms des GUI-Paketes. In diesem Ausschnitt sind die *MainFrame*, die *Faulttree*, die *EventVertexView* und die *CandVertexView* Klasse zu sehen. Die Klasse *Faulttree*, ist wie in dem Diagramm zu sehen, von der Klasse *JGraph* abgeleitet. Dadurch kann man diese Klasse in eine Java-GUI einbinden und sie bietet über bestimmte Methoden Schnittstellen an durch die man einfache Graphen Elemente darstellen kann. Damit die Fehlerbäume grafisch korrekt dargestellt

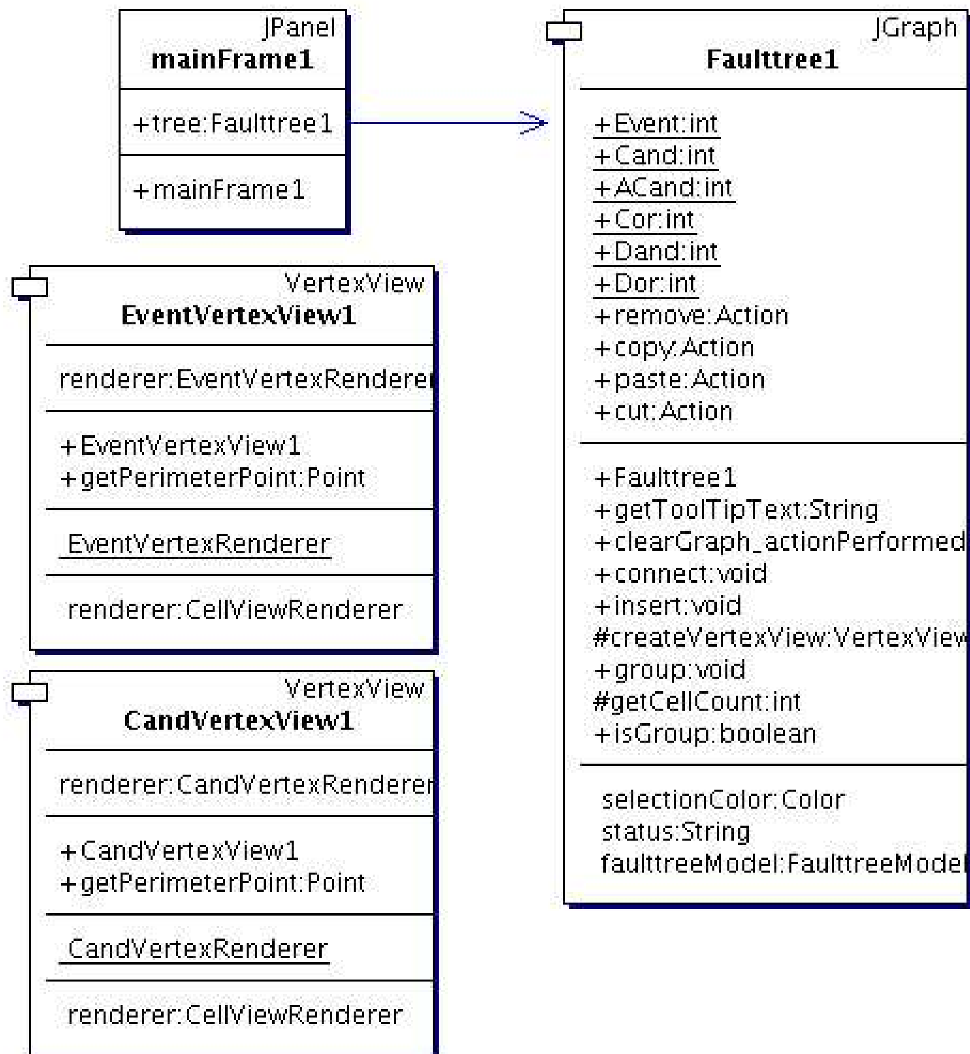

Abbildung 21: Teil des Klassendiagramms des GUI-Paketes

werden können, wurde zu jedem Gatter-Typ eine eigene GUI Klasse entwickelt, die von *VertexView* abgeleitet ist. In diesem Diagramm sind die Klassen *EventVertexView* und *CandVertexView* als Beispiele zu sehen. Diese Klassen stellen die Ansicht (engl: View) der einzelnen Gatter-Typen dar und enthalten als eingebettete Klasse jeweils einen *Renderer* mit einer *Paint*-Methode, die eine Grafik erzeugt, die den jeweiligen Gatter-Typ darstellt. Mit diesen View Klassen ist es möglich jeden Gattertyp grafisch unterschiedlich zu gestalten.
Weiterhin enthält das *GUI* Paket noch drei Unterpakete. Diese Pakete heißen *img*, *Listener_Handler* und *dialog*. In *img* sind alle verwendeten Bilder aus *Faulttree2Phaseautomata* untergebracht. *Listener_Handler* enthält Klassen die Methoden enthalten um auf bestimmte Ereignisse zu reagieren oder etwas zu überwachen. So ist hier z.B. eine Klasse zu finden, die die Tastatureingabe abfängt und dadurch das Löschen eines Objektes durch Drücken der „DEL“-Taste auslöst und andere Tastenkürzel umsetzt. Das Paket *dialog* beinhaltet Dialoge, wie den Dialog zum editieren eines Events. Diese Dialoge sind ähnlich der *MainFrame* Klasse aufgebaut und initialisieren ihre grafischen Komponenten in ihrem Konstruktor.

3.4.2 Entwurf der Logik

Das Java Paket in *Faulttree2Phaseautomata*, das die Logik-Schicht repräsentiert, heißt *control*. Dieses *control* Paket enthält die Anwendungslogik und alle Algorithmen die zur Umwandlung eines Fehlerbaumes in einen Phasenautomaten nötig sind.

Datenmodell der Fehlerbäume

Damit überhaupt ein Algorithmus auf einen gezeichneten Fehlerbaum angewendet werden, kann muss es ein Datenmodell des Fehlerbaumes geben, welches sich aus verschiedenen Java-Klassen zusammensetzt. Diese Klassen müssen die Knoten und die Kanten des Fehlerbaumes repäsentieren und durch Methoden alle Operationen, die auf einem Fehlerbaum möglich sind, anbieten.
Das Modell eines gezeichneten Fehlerbaumes wird implizit vom Benutzer aus Instanzen der Datenmodellklassen aufgebaut. Sobald ein Benutzer dem Fehlerbaum in der Benutzeroberfläche über einen Button ein Gatter hinzufügt wird dem Datenmodell des aktuellen Fehlerbaums ein neues Objekt hinzugefügt welches dieses neue Gatter repräsentiert. Da alle vorkommenden Gattertypen bekannt sind, existiert zu jedem Gatter-Typ eine eigene Klasse im *control* Paket. Abbildung 22 zeigt als Beispiele die Klassen die Event-Gatter und *AND-CC-Sync*- bzw. *Cand*-Gatter repräsentieren. Diese Klassen sind beide von der Klasse *FaulttreeVertexModel* und die wiederrum von der Klasse *TreeVertexModel* abgeleitet. In *FaulttreeVertexModel* sind Methoden und Eigenschaften gekapselt, die jeder Knoten in einem Fehlerbaum benötigt. In *TreeVertexModel* sind nur wenige allgemeine Eigenschaften enthalten, wie z.B. die automatische Vergabe einer neuen eindeutigen ID für jeden Knoten. Jede Klasse die einen Knoten in einem Fehlerbaum darstellt enthält die ID des Vorgängerknotens und die ID’s der Nachfolger. Dies erleichtert später die Implementierung der Algorithmen sehr stark.
Neben Klassen für Knoten enthält das *control* Paket auch Klassen für Kanten. Diese

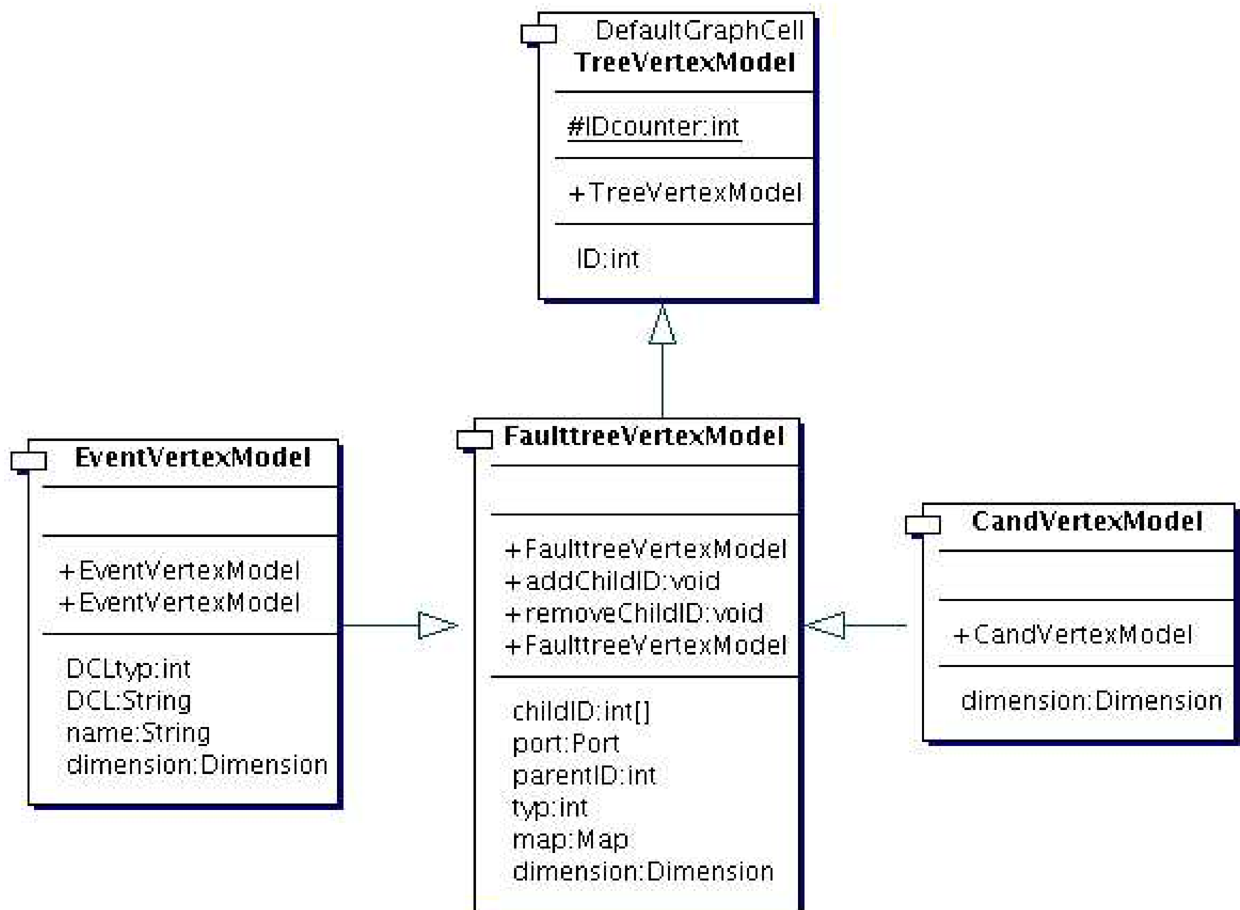

Abbildung 22: Klassen für Event- und AND-CC-Sync Gatter

Klassen sind in Abbildung 23 zu sehen. Diese Kantenklassen enthalten jeweils die ID ihres Ursprungs und ihres Zielknotens.
Abbildung 24 zeigt einen weiteren Teil des *control* Paketes. Dieses Klassendiagramm

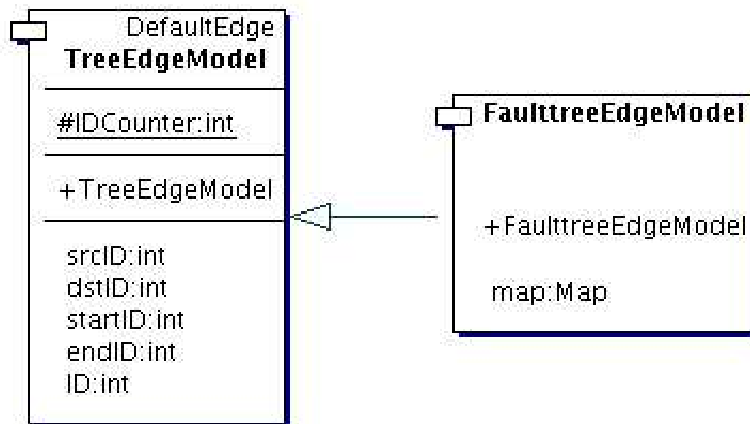

Abbildung 23: Klassen für Kanten eines Fehlerbaums

zeigt unter anderem die wichtigsten Klassen des Datenmodells eines Fehlerbaumes. Dies sind die Klassen *GraphModel*, *FaulttreeModel* und *FaulttreeParser*.
Die Klasse *GraphModel* ist eine Basisklasse. Sie enthält generelle Methoden um ein Datenmodell für einen Graphen zu erstellen, wie z.B. das Hinzufügen eines Knotens zu einem Graphen. Die Klasse *FaulttreeModel* ist von dieser Basisklasse abgeleitet und speziell darauf angepasst einen Fehlerbaum zu repräsentieren. Während in der *GraphModel* Klasse theoretisch eine Kante von jedem Knoten zu jedem Knoten gezogen werden kann, wird in der *FaulttreeModel* Klasse erst überprüft, ob eine Kante die gezogen werden soll, dem Standard eines Fehlerbaums entspricht. Ist dies der Fall, so wird diese Kante akzeptiert und gezogen. Ist dies nicht der Fall, so wird die Kante nicht gezogen. Die Überprüfung, ob eine Kante erlaubt ist oder nicht, erfolgt über die Klasse *FaulttreeParser*. Diese Klasse beinhaltet sozusagen den Standard [IEC93] für Fehlerbäume im Bezug auf den Aufbau des Baumes.
Die Knoten und Kantenobjekte, die von dem Benutzer dem Datenmodell hinzufügt werden, werden nach der Erstellung und Konfiguration ihrer Instanzen sofort in der Klasse *Faulttree* einem entsprechenden Vektor hinzugefügt, was dafür sorgen soll, das alle Elemente des Fehlerbaums ständig zur Verfügung stehen und nicht verloren gehen.

Datenmodell der Phasenautomaten

Das Datenmodell der Phasenautomaten besteht, ähnlich wie das Modell der Fehlerbäume, aus Knoten und Kanten. Im Unterschied zu den Fehlerbaummodellklassen sind hier jedoch andere Anpassungen erforderlich. Daher sind die Klassen *PhaseautomataStateModel* von *TreeVertexModel* und *PhaseautomataEdgeModel* von *TreeEdgeModel* abgeleitet. In Abbildung 25 ist ein Klassendiagramm der *PhaseautomataStateModel*, der *PhaseautomataVariable* und der *TreeVertexModel* Klasse zu sehen. Da in einem Phasenautomaten

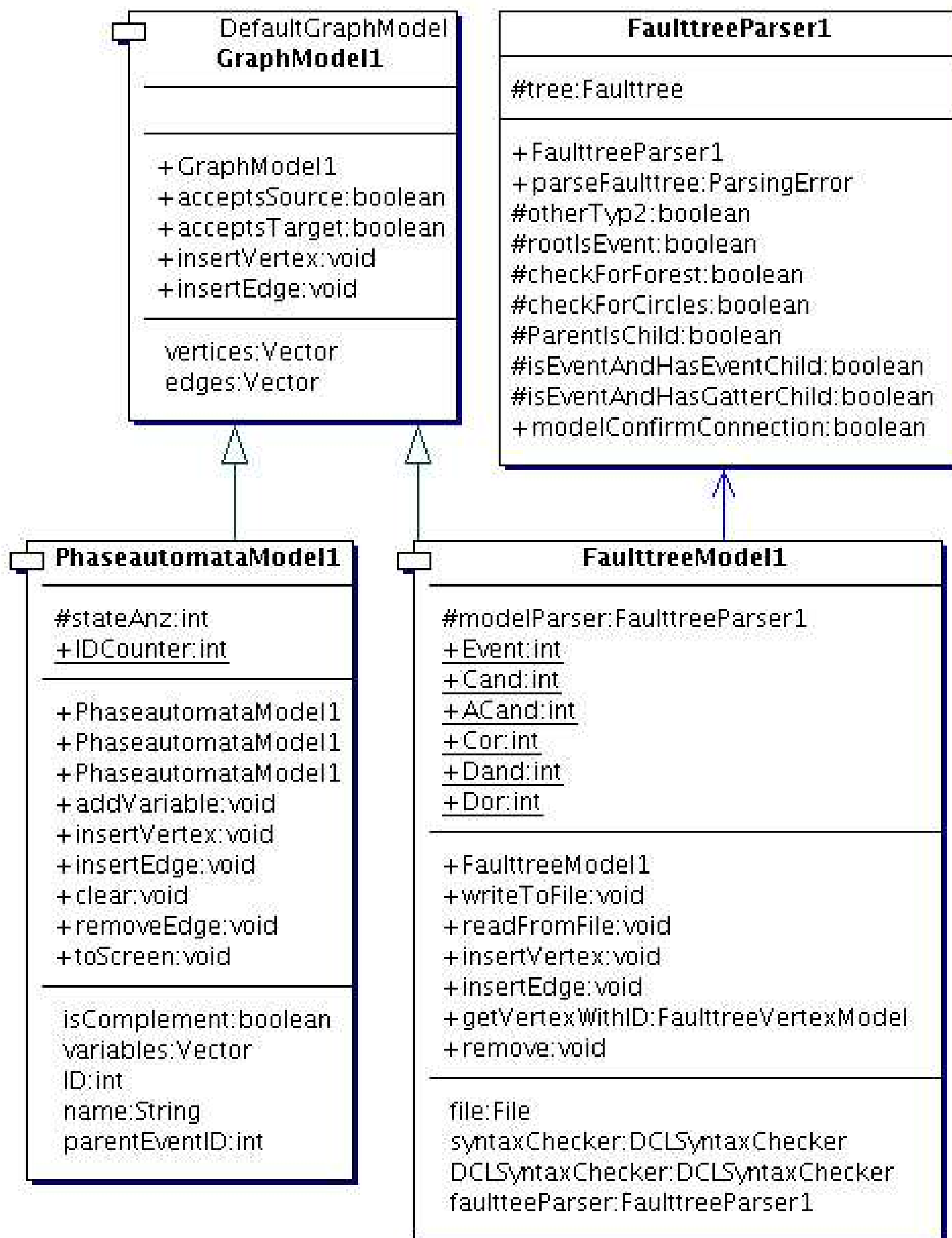

Abbildung 24: Teil des Klassendiagramms des Logik-Paketes

in jeder Phase ein bestimmter Ausdruck gilt, gibt es Variablen, die diese Ausdrücke als Werte annehmen können. Diese Variablen gehören jeweils zu den entsprechenden Phasen des Automaten.
Weiterhin gehören zu einer Phase eine oder mehrere Uhren, deren Modellklasse *PhaseautomataClock* ist. Diese Uhren werden in der zugehörigen Phase in einem Vektor gespeichert. Dies sind die wesentlichen Spezifikationen der Datenmodellklassen zu Phasenautomaten. Die Kantenklasse für Phasenautomaten ist der für Fehlerbäumen sehr

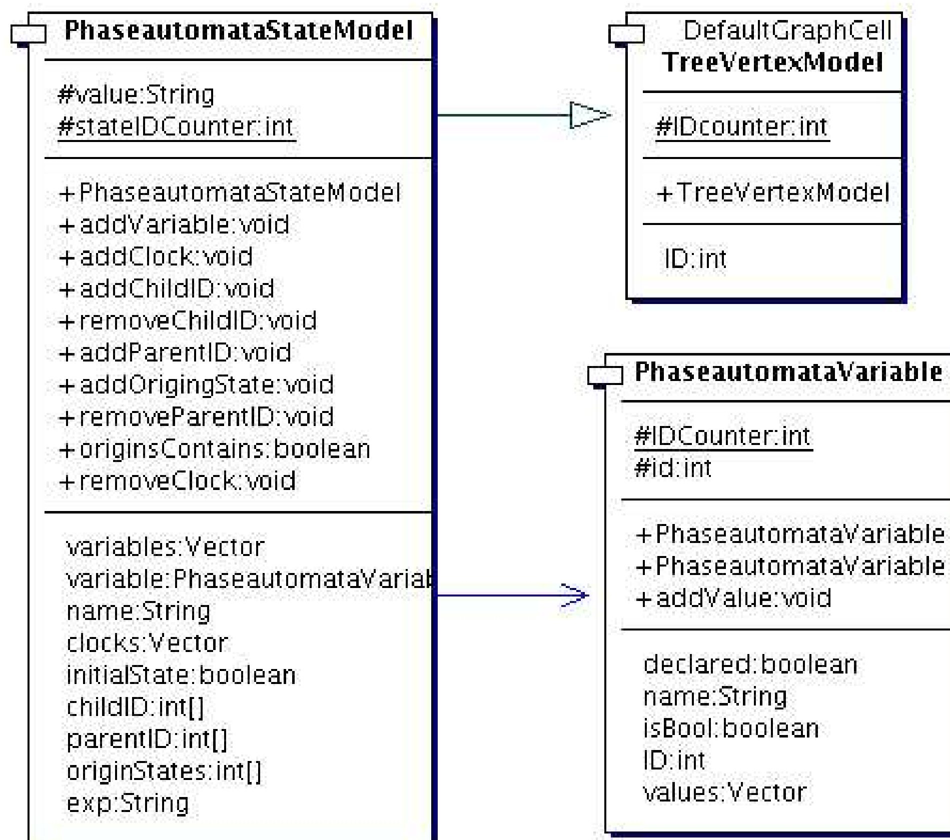

Abbildung 25: Datenmodellklassen der Phasen

ähnlich. Sie heißt *PhaseautomataEdgeModel*.

In Abbildung 26 ist ein Klassendiagramm der Klasse zu sehen, die die Uhren in Phasenautomaten modelliert. Zu jeder Phase können später eine oder mehrere dieser Uhren Klassen gehören. Diese Uhren werden in jeder Phase in einem Vektor gespeichert. Wie

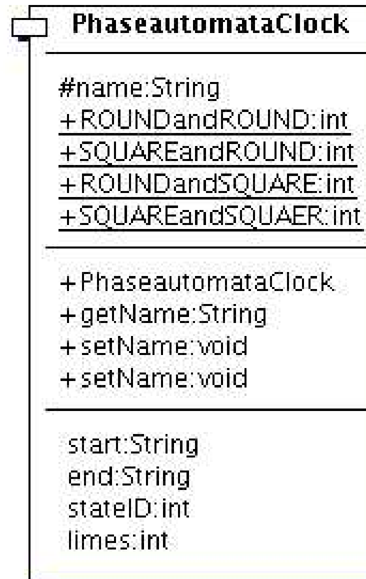

Abbildung 26: Datenmodellklassen der Uhren

bei den Fehlerbäumen gibt es auch bei den Phasenautomaten eine Klasse, die alle Knoten und Kanten die zu einem Automaten gehören zusammenfasst. Dies geschieht ebenfalls in Vektoren. Die Klasse, die diese Funktion übernimmt, heißt *PhaseautomataModel* und ihr Klassendiagramm ist in Abbildung 24 zu sehen. Diese Klasse bietet alle Funktionen die nötig sind um einen Phasenautomaten aus Instanzen der Kanten-, Variablen-, Clock- und Phasenklassen aufzubauen.

3.4.3 Export der Fehlerbäume in Phasenautomaten

Anders als bei den Fehlerbäumen kann der Benutzer die Phasenautomaten nicht direkt erstellen, sondern die Modelle werden vom Programm selber durch den Export erzeugt. Dieser Export kann vom Benutzer nur gestartet werden, wenn vorher sichergestellt worden ist, dass der Fehlerbaum der exportiert werden soll, formal korrekt gezeichnet worden ist. Also wird vor dem Export mit der Klasse *FaulttreeParser* (Abbildung 24) überprüft, ob der Fehlerbaum diese Bedingungen erfüllt. Wenn der Baum korrekt gezeichnet wurde, so wird über die Klasse *PhaseautomataExporter* der Export in das Dateiformat von MobyDC oder in ein XML Format ausgeführt. Die Klasse *PhaseautomataExporter* ist in Abbildung 27 zu sehen. In diesem Diagramm sind ebenfalls die Klassen *PhaseautomataProcessor* und *PhaseautomataFactory* zu sehen. Die Klasse *PhaseautomataProcessor* dient dazu aus bereits bestehen Phasenautomaten Modellen neue Phasenautomaten zu erstellen. Hier wurde unter Anderem eine Funktion zur disjunktiven Kombination von

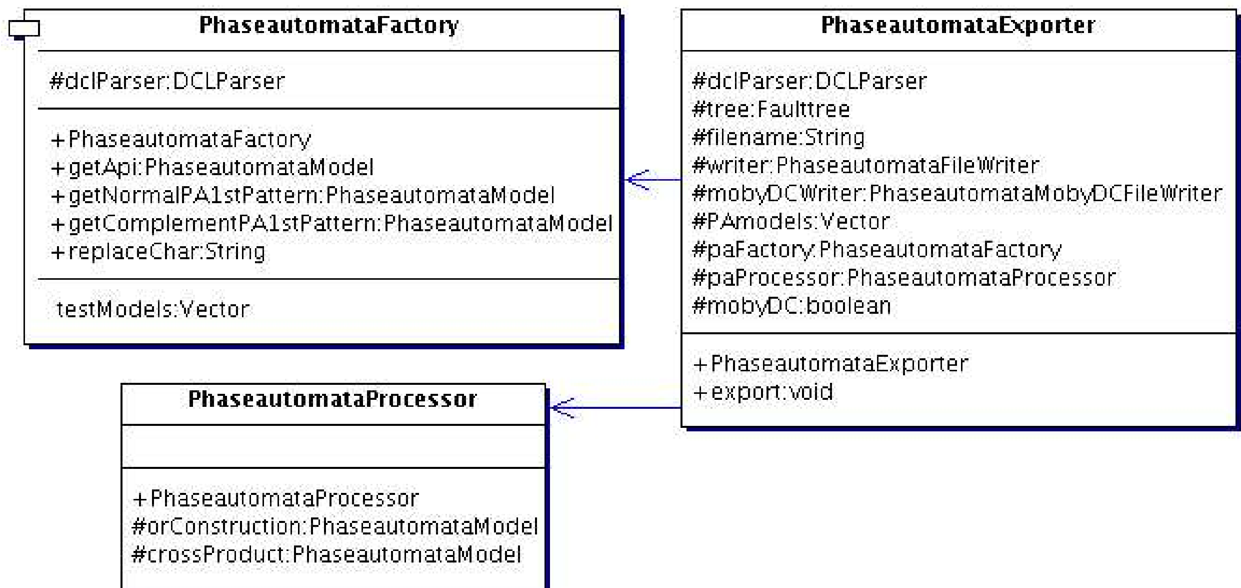

Abbildung 27: Klassen zum Export der Fehlerbäume

Phasenautomaten implementiert.
Die Klasse *PhaseautomataFactory* wird dazu verwendet neue Phasenautomatenmodelle zu erstellen und zu konfigurieren. Hier findet man Funktionen, die Phasenautomatenmodelle zu jedem der drei Pattern und deren Komplemente zurück liefert.

3.4.4 Datenhaltung

Nachdem ein Export durchgeführt worden ist, werden die entstandenen Phasenautomatenmodelle in Dateien geschrieben. Diese Dateien sind entweder in einem XML oder in einem von MobyDC lesbaren Format aufgebaut. Die Klassen, die diese Dateien schreiben sind in dem Unterpaket *control.fileWriting* enthalten. Hier finden wir ebenfalls Klassen zum Schreiben von Fehlerbäumen in ein XML Format. Zum Schreiben der XML Formate wird die Klasse *DOMWriter* aus der Standard-Bibliothek *org.w3c.dom* verwendet. Zum Schreiben des MobyDC Formats wurde eine eigene Klasse implementiert, die *PhaseautomataMobyDCFileWriter* heißt. In dieser Klasse wird über eine Methode ein *BufferedFileWriter* angesprochen, der dann alle nötigen Informationen über die zu speichernden Phasenautomaten in einem von MobyDC lesbaren Format in Dateien schreibt.

3.4.5 XML-Datenformat

Der Export von Phasenautomaten in ein XML-Datenformat wurde implementiert um die Umwandlung von Fehlerbäumen in Phasenautomaten auch für andere weiterverarbeitende Programme nutzbar zu machen. So könnte man z.B. ein anderes Programm als MobyDC zum Modelchecking verwenden, wenn dieses XML-Daten lesen kann. Es wird ebenfalls ein XML-Format verwendet um die gezeichneten Fehlerbäume abzuspeichern

und wieder zu laden. Die Struktur einer Phasenautomaten XML-Datei ist in Abbildung 28 zu sehen, die einen einzelnen Phasenautomaten zeigt. Die XML-Struktur der Phasenautomaten unterscheidet sich von der von Fehlerbäumen, da Phasenautomaten eine hirachische Struktur haben und Fehlerbäume nicht. Ein Beispiel eines in einer XML-Datei gespeicherten Fehlerbaumes ist in Abbildung 29 zu sehen. Im Format für Phasenauto-

```
<?xml version="1.0" encoding="UTF-8"?>
 <graph>
  <vertex type="phaseatomaton" id="1" name="K2 laenger als 60 ZE
  geschlossen">
   <graph>
    <vertex type="phaseautomaton.state" name="Pwait" id="1" exp="true"
    initial="true">
     <graph>
      <vertex type="phaseautomaton.clock" name="Cwait" start="0" end="w"
      limes="0"/>
     </graph>
    </vertex>
    <vertex type="phaseautomaton.state" name="Ppi" id="2" exp="k2_closed"
    initial="true">
     <graph>
      <vertex type="phaseautomaton.clock" name="Ck2_closed" start="60"
      end="w" limes="0"/>
     </graph>
    </vertex>
    <vertex type="phaseautomaton.state" name="Pfin" id="3" exp="true"
    initial="false"/>
    <edge type="phaseautomaton.edge" id="1" srcid="1" dstid="2"/>
    <edge type="phaseautomaton.edge" id="2" srcid="2" dstid="3"/>
   </graph>
  </vertex>
 </graph>
```

Abbildung 28: XML-Struktur von Phasenautomaten

maten kann jeder Knoten (vertex) von einem Graphen verfeinert werden. So kann eine Phase durch einen neuen Phasenautomaten verfeinert werden. Das Attribut *type* legt zu jedem Knoten fest, ob er z.B. eine Phase oder eine Uhr ist. Hierbei wurden sogenannte Namespaces verwendet. Wird z.B. einer Phase eine Uhr zugeordnet, so wird diese Uhr als Knoten in dem Graphen, der die ursprüngliche Phase verfeinert, gespeichert. Ein solcher Knoten kann aber theoretisch auch ein neuer Phasenautomat sein, wodurch Hirachie prinzipiell möglich ist, in *Faulttree2Phaseautomata* wird sie jedoch nicht benötigt. Die Kanten (edges) in einem Phasenautomaten können nicht verfeinert werden und sind daher auch nicht durch einen Graphen verfeinert sondern werden am Ende der XML-

Datei mit den IDs der Anfangs- und der Endphase gespeichert. Die Dateiendung für Phasenautomaten ist .pax und steht für **P**hase**a**utomata **X**ML Hier werden zunächst alle Knoten (vertices) mit den dazugehörigen inhaltlichen und geometrischen Attributen gespeichert. Im zweiten Teil finden sich alle Kanten (edges) wieder, die als Attribute jeweils die ID des Start und des Ziel Knotens enthalten.

```
<?xml version="1.0" encoding="UTF-8"?>
 <graph>
  <vertices>
   <vertex>
    <faulttreeVertexAttr id="1"
    typ="0" name="K2 oeffnet nicht als kein EMF anliegt"
    dcl="[k2.closed and notk2.EMF]" dcl_typ="1"/>
    <vertexGeo width="221" height="35" x="240" y="130"/>
   </vertex>
   <vertex>
    <faulttreeVertexAttr id="2" typ="0"
    name="K2 laenger als 60 ZE geschlossen"
    dcl="[k2.closed]&gt;60" dcl_typ="1"/>
    <vertexGeo width="201" height="35" x="100" y="20"/>
   </vertex>
   <vertex>
    <faulttreeVertexAttr id="3" typ="0"
    name="EMF an K2 laenger als 60 ZE" dcl="[k2.EMF]&gt;60"
    dcl_typ="1"/>
    <vertexGeo width="181" height="35" x="40" y="130"/>
   </vertex>
   <vertex>
    <faulttreeVertexAttr id="4" typ="5" name="other"
    dcl="other" dcl_typ="-1"/>
    <vertexGeo width="32" height="32" x="190" y="80"/>
   </vertex>
 </vertices>
 <edges>
  <edge id="1">
   <edgeAttr srcid="2" destid="4"/>
  </edge>
  <edge id="2">
   <edgeAttr srcid="4" destid="3"/>
  </edge>
  <edge id="3">
   <edgeAttr srcid="4" destid="1"/>
  </edge>
 </edges>
</graph>
```

Abbildung 29: XML-Struktur von Fehlerbäumen

4 Fallbeispiel

Es soll nun der praktische Umgang mit *Faulttree2Phaseautomata* mit Hilfe eines Beispiels gezeigt werden. Es wird hier ein Teil des *Pressure-Tank* Modells aus [VGRH81] verwendet.

4.1 Beschreibung des Systems

Die Beschreibung des in Abbildung 30 dargestellten Pressure-Tank Systems wurde aus [Sch02] übernommen.

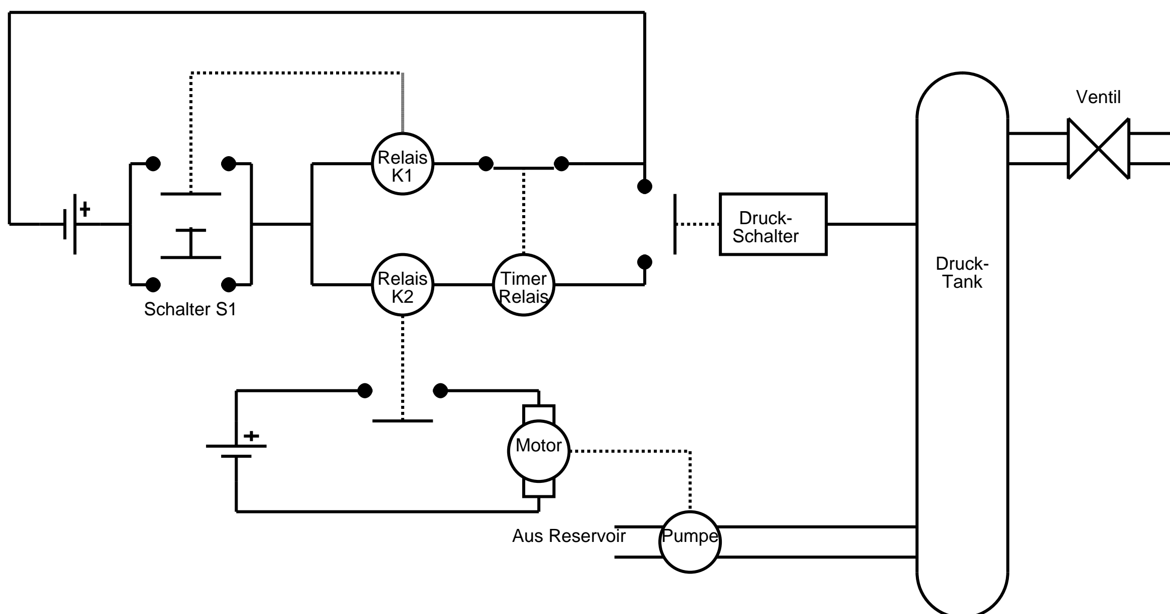

Abbildung 30: Schaltbild des Pressure-Tank Beispiels

Der Drucktank wird durch die Pumpe innerhalb von 60 Sekunden unter Druck gesetzt und durch das Ventil in vernachlässigbarer Zeit wieder geleert. Es wird angenommen, dass die Pumpe die Flüssigkeit aus einem unendlich großen Reservoir bezieht. Solange der Drucktank noch nicht vollständig gefüllt ist, sind die Kontakte des Druck-Schalters geschlossen und sobald der gewünschte Druck erreicht ist, öffnen sie sich. Dadurch wird die Spule von Relais K2 vom Strom getrennt, das Relais öffnet sich und die Pumpe hört auf zu arbeiten. Wenn der Tank geleert ist, schließen sich die Kontakte des Druckschalters wieder und der Zyklus beginnt erneut.
Initial befindet sich das System im Ruhezustand:

- Die Kontakte des Schalters S1 sind offen.
- Die Kontakte von Relais K1 und Relais K2 sind offen.
- Es fließt kein Strom durch das Kontrollsystem.

- Die Kontakte des Timer-Relais sind geschlossen.
- Der Tank ist leer und deshalb sind die Kontakte des Druckschalters ebenfalls geschlossen.

Das System wird durch kurzzeitiges Schließen von Schalter S1 gestartet. Dadurch wird der Stromkreis geschlossen, die Spule von K1 aktiviert und das Relais K1 geschlossen. Dadurch sorgt K1 selbst dafür, dass seine Spule mit Strom versorgt wird. Außerdem wird auch die Spule von K2 unter Strom gesetzt, so dass K2 schließt und die Pumpe anfängt zu arbeiten.
Das Timer-Relais dient als redundantes System, falls der Druck-Schalter ausfällt. Sobald das Relais 60 Sekunden kontinuierlichen Stromfluß misst, werden die Kontakte des Relais geöffnet, die Spule von K1 vom Strom getrennt. Damit öffnen sich die Kontakte von K1 und der Strom an K2 wird abgeschaltet, so dass die Pumpe stoppt.
Für die Analyse wird in [VGRH81] angenommen, dass der Tank in jedem Fall berstet, wenn die Pumpe länger als 60 Sekunden arbeitet.

4.2 Fehlerbaum-Analyse

In dieser Arbeit soll zur Demonstration der Arbeitsweise von *Faulttree2Phaseautomata* nur ein Teil des Fehlerbaums zum *Pressure-Tank* Beispiel auf Korrektheit und Vollständigkeit überprüft werden. Als Teilbereich untersuchen wir das Ereigniss „K2 länger als 60 ZE geschlossen“ welches durch die Ereignisse „EMF an K2 länger als 60 ZE “ und „K2 öffnet nicht als kein EMF anliegt “ ausgelöst wird, die über ein OR Gatter verbunden sind. Das bedeutet eines der beiden Ereignisse muss eintreten, damit das übergeordnete Ereignis eintritt.
In [VGRH81] und [Thu01] wurde eine Fehlerbaumanalyse des Systems für das Ereignis „K2 länger als 60 ZE geschlossen“ durchgeführt. Es wird hier die Analyse aus [Thu01] übernommen. Zusätzlich wird hier jedes Ereignis formal durch eine DCL-Formel beschrieben. Abbildung 31 zeigt den Teil des Fehlerbaums aus [Thu01], der hier untersucht werden soll. Wir bezeichnen im folgenden das Ereignis „K2 länger als 60 ZE geschlossen“ mit **E3** und das Ereignis „EMF an K2 länger als 60 ZE“ mit **E31**, sowie das Ereignis „K2 öffnet nicht als kein EMF anliegt“ mit **E32** .

4.3 Modellierung des Systems

Zur Untersuchung dieses Abschnittes benötigen wird eine formales Modell des Systems *Pressure-Tank*, wobei wir hier nicht das komplette System modellieren sondern nur den benötigten Teil. Eine komplette Modellierung findet sich in [Sch02]. Das Modell, dass das Verhalten des Systems beschreibt, besteht aus einer Menge von Phasenautomaten, welche später gemeinsam mit den Phasenautomaten, die von *Faulttree2Phaseautomata* generiert werden, auf Leerheit getestet werden.
Für den gewählten Teilbereich benötigen wir nur die Modellierung des Relais K2. Der Phasenautomat in Abbildung 32 modellieren das Verhalten des Relais K2. Ob das Relais offen oder geschlossen ist, wird durch die Observable `k2` : $(k2_open, k2_close)$ und

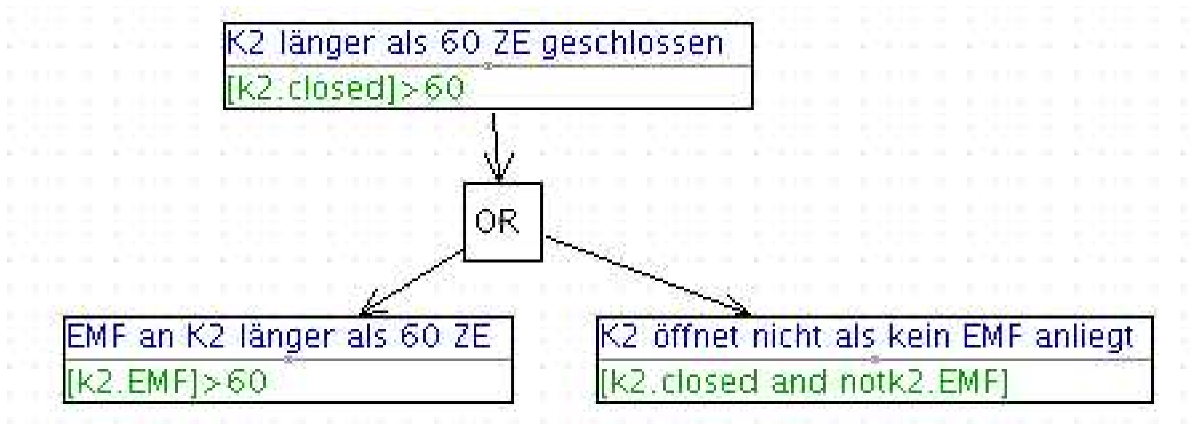

Abbildung 31: Teil des Fehlerbaums zum Beispiel Pressure Tank

den Phasenautomaten **k2** bestimmt. Das Relais kann auch ausfallen, d.h. obwohl kein elektromagnetisches Feld an der Spule anliegt, verbleibt das Relais in geschlossenem Zustand. Aus diesem Grund wird zusätzlich die Observable k2power: $(k2_EMF, k2_noEMF)$ verwendet, die beschreibt, ob ein elektromagnetische Feld (EMF) an der Spule des Relais anliegt oder nicht.
Die Modellierung des benötigten Systems muss vom Benutzer von Hand vorgenommen werden. Dazu werden Phasenautomaten in MobyDC eingegeben. Die später von *Faulttree2Phaseautomata* generierten Phasenautomaten können an die entstandene MobyDC-Projekt-Datei angehängt werden.

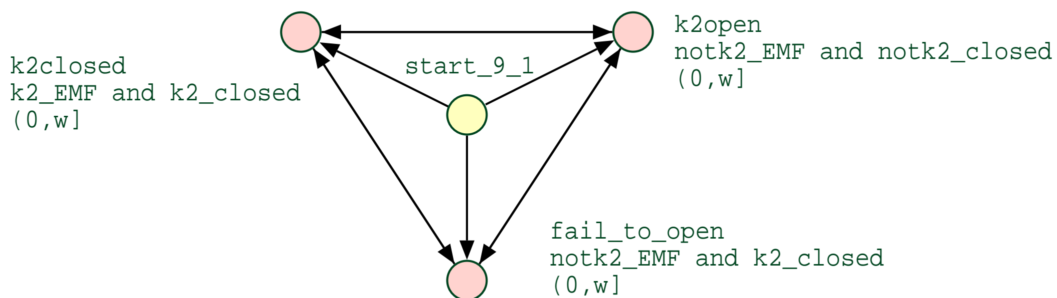

Abbildung 32: Modellierung des Relais K2

4.4 Modelchecking der Phasenautomaten mit MobyDC

Um die Korrektheit und die Vollständigkeit des Fehlerbaums zu zeigen müssen wir nun den in *Faulttree2Phaseautomata* gezeichneten Fehlerbaum durch die Export Funktion exportieren und in MobyDC laden.

In [Sch02] wird das Vorgehen beim Beweisen der Korrektheit und Vollständigkeit detailliert beschrieben.

4.4.1 Beweis der Korrektheit

Zum Beweis der Korrektheit müssen wir

$$\mathbf{neg_E3} \| \mathbf{E31_or_E32} \| \mathbf{K2}$$

auf Leerheit prüfen. Damit wäre die Implikation

$$(\Diamond \lceil k2_EMF \rceil^{>60} \vee \Diamond \lceil k2_closed \wedge \neg k2_EMF \rceil) \Rightarrow \Diamond \lceil k2_closed \rceil^{>60}$$

für diese Modellierung bewiesen.
Der Phasenautomat **k2** wird in Abbildung 32 gezeigt. Der Automat **E31_or_E32** wird von *Faulttree2Phaseautomata* generiert und wird in Abbildung 33 gezeigt sowie auch

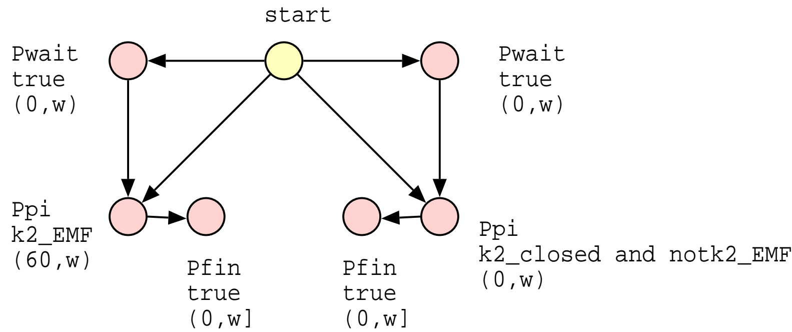

Abbildung 33: Phasenautomat **E31_or_E32**

der Automat **neg_E3** von *Faultree2Phaseautomat* generiert wird und in Abbildung 34 zu sehen ist. Der Modellchecker in MobyDC gab für die Automaten

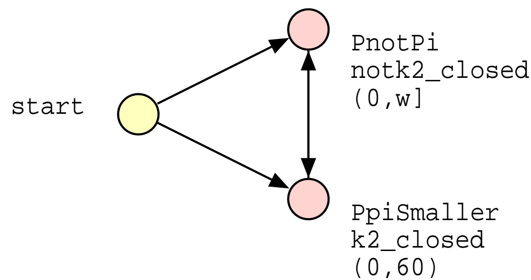

Abbildung 34: Phasenautomat **neg_E3**

$$\mathbf{neg_E3} \| \mathbf{E31_or_E32} \| \mathbf{K2}$$

den Dialog aus, der in Abbildung 35 zu sehen ist. Das TRUE, das am rechten unteren Rand angezeigt wird, heißt in diesem Fall, das kein gemeinsamer Ablauf existiert und amit die Korrektheit bewiesen ist.

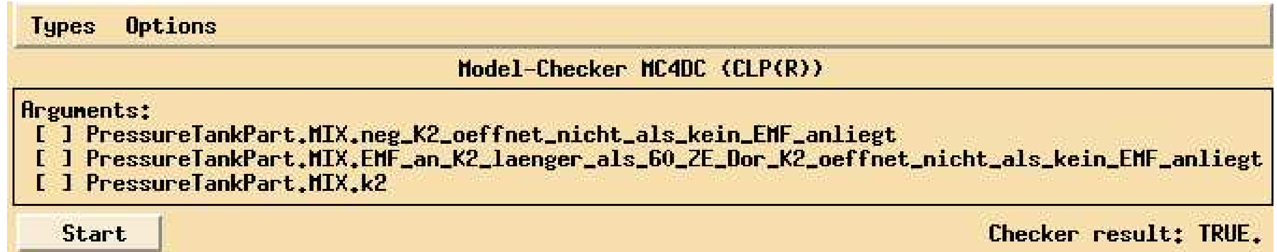

Abbildung 35: Ergebnisdialog des Modellcheckings in MobyDC

4.4.2 Beweis der Vollständigkeit

Die Vollständigkeitsbedingung

$$\Diamond\lceil k2_closed \rceil^{>60} \Rightarrow (\Diamond\lceil k2_EMF \rceil^{>60} \vee \Diamond\lceil k2_closed \wedge \neg k2_EMF \rceil)$$

kann durch den Leerheitstest von

$$\mathbf{E3} \| \mathbf{neg_E31} \| \mathbf{neg_E32} \| \mathbf{K2}$$

bewiesen werden.
Die Automaten **E3**, **neg_E31** und **neg_E32** werden von *Faulttree2Phaseautomata* erstellt und sind in den Abbildungen 36, 37 und 38 gezeigt. Der Automat **k2** gehört zur Modellierung des Systems und ist in Abbildung 32 zu sehen. Das Modelchecking in MobyDC ergibt auch hier, dass kein gemeinsamer Ablauf existiert, wie in Abbildung 39 zu sehen ist. Somit ist auch die Vollständigkeit bewiesen und der untersuchte Teil des Fehlerbaums ist daher korrekt und vollständig.

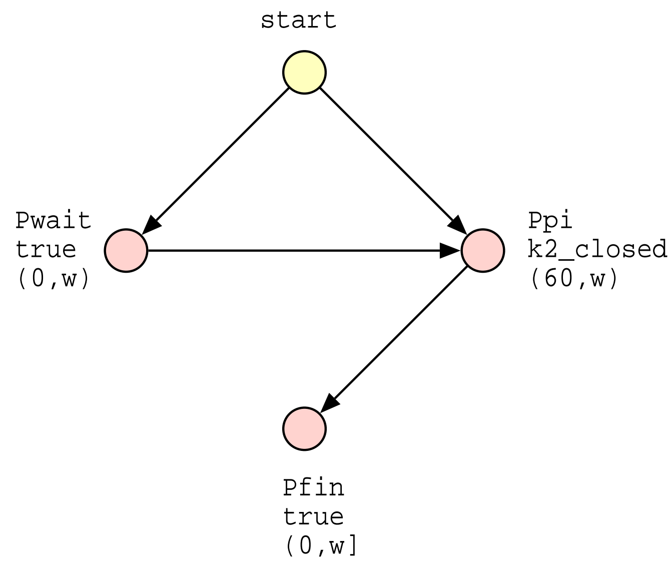

Abbildung 36: Phasenautomat **E3** (K2 länger als 60 ZE geschlossen)

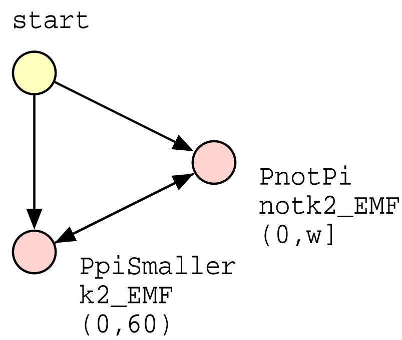

Abbildung 37: Phasenautomat **neg_E31**

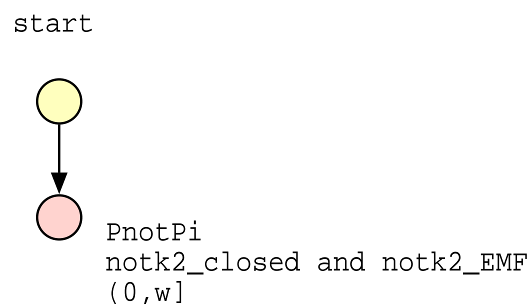

Abbildung 38: Phasenautomat **neg_E32**

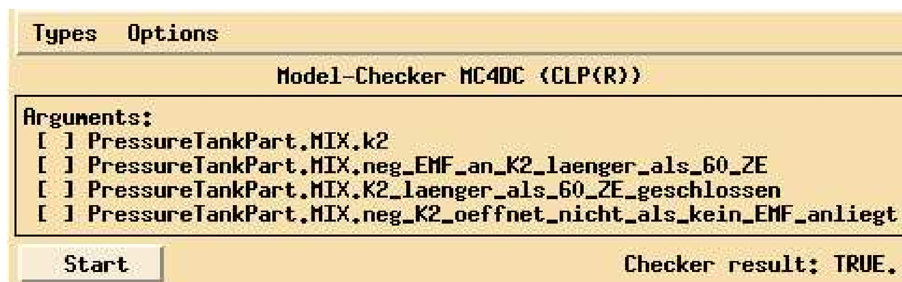

Abbildung 39: Ergebnisdialog des Modellcheckings in MobyDC

5 Ausblick

Es ist mit *Faulttree2Phaseautomata* erreicht worden, die Fehlerbaumanalyse zu vereinfachen, da auch ein nicht hochqualifizierter Benutzer mit diesem Tool arbeiten kann. Das Verständniss aller Mechanismen des Model-Checkings ist nicht mehr notwendig um einen Fehlerbaum zu verifizieren.
Alleine für sich gesehen ist *Faulttree2Phaseautomata* ein Werkzeug, dass dazu dienen kann, Fehlerbäume zu erstellen, zu bearbeiten und zu archivieren. Nur in Verbindung mit einem Model-Checker ist es in der Praxis zur formalen Verifikation nutzbar. Durch den zusätzlichen Export in ein XML Format ist *Faulttree2Phaseautomata* dafür gerüstet auch mit anderen Model-Checkern als MobyDC zusammen zu arbeiten.
Als Erweiterung könnte eine kontextbezogene Hilfe dem Benutzer den intuitiven Umgang mit *Faulttree2Phaseautomata* noch weiter erleichtern.
Außerdem wäre es denkbar den Export in Phasenautomaten noch weiter auszubauen. In dieser Arbeit ist bei der Komplement Konstruktion des dritten Patterns nur eine bestimmte Teilklasse von Ereignissen berücksichtigt worden. Es wäre durch eine andere Art von Konstruktion eventuell möglich diese Teilklasse zu erweitern. Als nützlicher Zusatz könnte auch ein Export in ein graphisches Format wie *.eps* implementiert werden, um Fehlerbäume in Dokumenten einfach darstellen zu können.

6 Literatur

Literatur

[Bec83] Becker, A.: *Automatische Fehlerbaumerstellung*. Doktorarbeit, Technische Universität Berlin, 1983.

[CHR91] C. Zhou, C.A.R. Hoare und A. P. Ravn: *A calculus of durations*. Information Processing Letters, 40(5):269–276, 1991.

[Gór94] Górski, J.: *Extending Safety Analysis Techniques with Formal Semantics*. In: Redmill, Felix (Herausgeber): *Technology and assessment of safety-critical systems : proceedings of the Second Safety-Critical Systems Symposium*, Seiten 147–163. Springer Verlag Berlin, 1994.

[IEC93] *DIN IEC 61025: Störungsbaumanalyse*, 1993.

[JFl] JFlex. http://www.jflex.com.

[Jgr] Jgraph. http://www.jgraph.com.

[JLe] JLex. http://www.jlex.com.

[RST00] Reif, W., G. Schellhorn und A. Thums: *Safety Analysis of a Radio-Based Crossing Control System Using Formal Methods*. In: *Proceedings of the 9th IFAC Symposium Control in Transportation Systems 2000 June 13-15, Braunschweig, Germany*, 2000.

[RST01] Reif, W., G. Schellhorn und A. Thums: *Integration formaler Spezifikationen und Sicherheitsanalyse*. Technischer Bericht 2001-6, Institut für Informatik, Universität Augsburg, Juni 2001.

[RST02] Reif, W., G. Schellhorn und A. Thums: *Formal Fault Tree Semantics*. Technischer Bericht, Institut für Informatik, Universität Augsburg, 2002.

[Sch02] Schäfer, A.: *Fehlerbaumanalyse und Model-Checking*. Diplomarbeit, University of Oldenburg, 2002. in German.

[Ska94] Skakkebæk, J. U.: *Liveness and Fairness in Duration Calculus*. In: Jonsson, B. und J. Parrow (Herausgeber): *CONCUR‘94*, volume 836 of Lecture Notes in Computer Science, Seiten 283–298. Springer-Verlag, 1994.

[Sun] Sun Microsystems. http://java.sun.com.

[Tap01] Tapken, J.: *Model-Checking of Duration Calculus Specifications*. Doktorarbeit, Carl von Ossietzky Universität Oldenburg, 2001.

[Thu01] Thums, A.: *The Pressure Tank Example*. Universität Augsburg, September 2001.

[VGRH81] Veseley, W.E., F.F. Goldberg, N.H. Roberts und D.F. Haasl: *Fault Tree Handbook*. Washington DC: US Nuclear Regulatory Commission, NUREG-0492, 1981.

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.