Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Studienarbeit

Erweiterung von Syspect um einen Editor für Phasen-Event-Automaten

Matthias Peters

12.10.2009

Prüfer: Prof. Dr. E.-R. Olderog
Betreuer: Dipl.-Inform. Johannes Faber

Zusammenfassung

Das UML-Modellierungswerkzeug Syspect bietet die Möglichkeit, Spezifikationen durch den Model-Checker ARMC automatisch zu verifizieren. Für den Export in Transition-Constraint-Systeme, der Eingabesprache für ARMC, werden zwischenzeitlich interne Phasen-Event-Automaten (PEAs) auf Basis des PEA Toolkits erzeugt. Der als neues Syspect-Plugin entworfene PEA-Editor ermöglicht die Generierung von Phasen-Event-Automaten und deren Visualisierung und Bearbeitung, wobei sich der Editor für den Benutzer intuitiv verständlich in Syspect integriert. Bei dem Entwurf wurden möglichst viele Klassen aus den anderen Syspect-Plugins wiederverwendet oder erweitert, um Redundanzen zu vermeiden. Das SyspectPEAEditor-Plugin besteht überwiegend aus zwei Teilen, dem grafischen Teil zur Anzeige und Bearbeitung von PEAs und dem logischen Teil, der sich um die Konvertierungen zwischen den verschiedenen PEA Versionen, bzw. um weitere Importe und Exporte kümmert.

Inhaltsverzeichnis

1. Einleitung **6**

2. Grundlagen **7**
2.1. CSP-OZ-DC 7
2.1.1. CSP 7
2.1.2. OZ 8
2.1.3. DC 9
2.1.4. CSP-OZ-DC-Klassen 12
2.2. UML-Profil für CSP-OZ-DC 12
2.3. Phase-Event-Automaten als Semantik von CSP-OZ-DC 15
2.3.1. Phasen-Event-Automaten 15
2.3.2. Übersetzung von CSP-OZ-DC in Phasen-Event-Automaten 16
2.4. Eclipse 17
2.4.1. Plugins 18
2.4.2. RCP - Rich Client Platform 18
2.5. GEF - Graphical Editing Framework 18
2.6. SWT und JFace 19
2.7. Syspect 19
2.7.1. PEA-Toolkit 19
2.7.2. Community-Z-Tools 19

3. Anforderungen **21**
3.1. Bestandsanalyse: PEAs in Syspect 21
3.1.1. MobyPEA 21
3.2. Vorgaben an den Syspect PEAEditor 21
3.2.1. PEA-Projekte 22
3.2.2. Sammlung von PEAs 23
3.3. Anwendungsbeispiel ”Elevator” 23

4. Entwurf und Implementierung **29**
4.1. Verwendete Teile von Syspect 29
4.2. Einbettung von *PEAEditor* in Syspect 32
4.2.1. Integration des PEA-Modells in das Syspect-Modell 33
4.2.2. Verwendete Extensions 33
4.3. Das PEAEditor-Plugin 35
4.3.1. Paket-Struktur des PEAEditors-Plugins 35
4.3.2. Der Editor für Phasen-Event-Automaten 35
4.3.3. Die Konverter 39

5. Zusammenfassung und Ausblick **46**

A. Pakete und Klassen **50**
A.1. Plugin SyspectPEAEditor 50
A.2. sonstige Syspect-Plugins 53

Abbildungsverzeichnis

1. Z-Schema *Fahrstuhl* 9
2. Dekoriertes Z-Schema *Fahrstuhl* 9
3. Z-Schema der Operation *fahreZu* 10
4. Z-Schema der Operation *fahreZu* mit Δ-Operator 10
5. Syspect-Klassendiagramm mit TabbedPropertyView 24
6. Syspect-Statemachine 24
7. DC-Specifikation 25
8. CSP-OZ-DC-Specification „Elevator“ 26
9. PEA-XML-Export 27
10. Moby/PEA: PEA-System 27
11. Moby/PEA: Verfeinerung CSP-Elevator 28
12. Die Paketstruktur 35
13. VPhaseLabelImpl 36
14. Die Klasse AttributeSection 37
15. Editierbare AttributeSection 38
16. PEALayoutAction 39
17. Eingliederung der Syspect-PEAs 40
18. Export einer Syspect-Spezifikation (über das Zwischenformat PTK-PEA) in Syspect-PEA 41
19. Generierter Syspect-PEA 43
20. Export von Syspect-PEAs nach PEA-XML über das Zwischenformat PTK-PEA 44
21. Wizards für Exporte von Syspect-PEAs 45

1. Einleitung

Eine wichtige Anforderung an Software ist deren Korrektheit. Es ist wünschenswert, dass schon bei dem Entwicklungsprozess eine Verifikation der Software stattfindet, indem die Eigenschaften eines in einer Modellierungssprache wie UML [RJB99, OMG03a, OMG03b] erstellten Modells überprüft werden. Syspect ist die Implementierung eines Konzepts, die formalen Spezifikationen in den Software-Engineering-Prozess zu integrieren. Es wird hierbei ein spezielles UML-Profil [MORW08] verwendet, bei dem die UML-Elemente über eine durch CSP-OZ-DC [HO02a, HO02b] definierte Semantik verfügen. CSP-OZ-DC ist eine Kombination von Spezifikationstechniken für Prozesse, Daten und Realzeitanforderungen, die eine Transformation in Phasen-Event-Automaten (PEA) [Hoe06] erlauben. Ein PEA ist ein Automat, der Konzepte für nebenläufige und kommunizierende System, für Realzeit und für Daten umfasst. Solche PEAs werden dann im weiteren Verlauf in Transition-Constraint-Systems (TCS) umgewandelt, die wiederum für die Verifikation durch einen Model-Checker, wie z.B den Abstraction-Refinement-Model-Checker ARMC, verwendet werden. ARMC erhält als Eingabe einen unerwünschten Zustand und zeigt, dass dieser in der Spezifikation nicht auftreten kann. Tritt der unerwünschte Zustand aber auf, so liefert ARMC den Ablauf bis zu diesem Zustand als Gegenbeispiel für die zu untersuchende Eigenschaft.

In dieser Studienarbeit geht es um eine Erweiterung der Modellierungsumgebung Syspect. Als Zwischenprodukt vom UML-Design zur ARMC-Verifikation treten die Phasen-Event-Automaten in Syspect nur temporär auf. Sie können allerdings in einem speziellen XML-Format abgespeichert und in externen Tools weiterbearbeitet werden. Dies soll ein besseres Verständnis, Optimierungen und die Fehlersuche unterstützen. Momentan steht hierzu das Moby/PEA zur Verfügung, welches aber zahlreiche Inkompatibilitäten zu Syspect enthält. Eine bessere Alternative wäre es aber, die PEA innerhalb des Syspect-Systems visualisieren und editieren zu können. Genau dieses Ziel soll diese Studienarbeit verfolgen: Dem Benutzer von Syspect soll die Möglichkeit gegeben werden, direkt zu einem UML-Element einen PEA zu erzeugen bzw. PEAs zum Gesamt-/ Teilsystem. Anschließend kann der Benutzer die PEAs dann in der Syspect-Modellierumgebung weiter bearbeiten.

2. Grundlagen

Dieser Abschnitt gliedert sich in zwei Teile. Zunächst wird ein Blick auf den theoretischen Hintergrund der Spezifikations-Umgebung Syspect und die zu entwerfende Syspect-Erweiterung für Phase-Event-Automaten geworfen. Dieser Bereich umfasst die Beschreibung der Spezifikationssprache CSP-OZ-DC, die Erweiterung der UML um entsprechende Elemente, die eine eindeutige Übersetzung von UML-Diagrammen in CSP-OZ-DC ermöglichen und schließlich die Umwandlung von CSP-OZ-DC in Phasen-Event-Automaten, die eine automatische Verifikation erlauben. Der andere Teil beschäftigt sich mit den programmierspezifischen Grundlagen (Entwicklungsumgebung, zusätzliche Pakete).

2.1. CSP-OZ-DC

Die Spezifikationssprache CSP-OZ-DC ist die Kombination folgender Spezifikationssprachen: CSP (Communicating Sequential Processes) zur Beschreibung von kommunizierenden Prozessen, OZ (Object-Z) als objektorientierte Version der Spezifikationssprache Z zur Beschreibung von Daten und Zustandsräumen und DC (Duration Calculus) zur Darstellung der zeitlichen Eigenschaften von Systemen. Damit sich später in einfacher Weise Model-Checking bei einem mit CSP-OZ-DC spezifiziertem System durchführen lässt, müssen die einzelnen Teile über eine gemeinsame Semantik verfügen. Diese wurde in der in Form von Phasen-Event-Automaten entwickelt. In den folgenden Unter-Kapiteln gibt es eine kurze Erläuterung zu den einzelnen CSP-OZ-DC-Teilen und zu den Grundblöcken der Spezifikation, den CSP-OZ-DC-Klassen. Ausführlich werden diese Punkte in der Dissertation von Jochen Hoenicke [Hoe06] behandelt. Aus dieser Arbeit stammt auch das „Elevator“-Beispiel, das auch hier in späteren Kapiteln verwendet wird.

2.1.1. CSP

Die Spezifikationssprache CSP wurde 1978 von Hoare [Hoa78, Hoa85] eingeführt. Sie beschreibt die Ausführung sequentieller und paralleler Prozesse und ihre Kommunikation über Ereignis-Kanäle. Das Kommunikations-Ereignis ist als (c, v) strukturiert, wobei c ein Kommunikationskanal und v der übertragene Wert ist. Die Menge aller möglichen Kommunikations-Ereignisse eines Prozesses bildet sein Alphabet A. Die Kommunikation findet augenblicklich und synchron zwischen parallelen Prozessen statt. Die Syntax von CSP lässt sich mit einer einfachen Grammatik beschreiben:

$$P ::= \mathtt{Stop} \mid \mathtt{Skip} \mid a \rightarrow P \mid P_1 \sqcap P_2 \mid P_1 \square P_2 \mid P_1 \fatsemi P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid P \setminus A \mid P[R] \mid X$$

`Stop` Der Prozess nimmt an keiner Kommunikation mehr teil (Deadlock).

`Skip` Der Prozess terminiert auf der Stelle. Das Terminierungs-Ereignis ✓wird dabei kommuniziert.

$a \rightarrow P$ Ein Ereignis a wird kommuniziert und anschließend verhält sich der Prozess wie P.

$P_1 \sqcap P_2$ Das Konstrukt verhält sich entweder wie P_1 oder wie P_2, wobei die Wahl intern getroffen wird.

$P_1 \,\Box\, P_2$ Die Wahl, ob P_1 oder P_2 ausgeführt wird, wird extern von einer parallel laufenden Komponente getroffen.

$P_1 \,⨾\, P_2$ Sequentielle Komposition: Ein von Prozess P_1 kommuniziertes Terminierungs-Ereignis $\checkmark$ sorgt für die Aktivierung von P_2.

$P_1 \underset{A}{\|} P_2$ Parallele Komposition: Die zwei Prozesse werden synchronisiert über dem Alphabet A ausgeführt, welches eine Menge von gemeinsamen Ereignissen von P_1 und P_2 ist. Nur wenn beide Prozesse ein Ereignis aus dieser Menge kommunizieren wird eine simultane Transition ausgeführt. Andere Ereignisse, die nicht im Alphabet A sind, kann jeder Prozess für sich kommunizieren.

$P \setminus A$ Hiding: Die Ereignisse aus A werden versteckt.

$P[R]$ Umbenennen der Ereignisse durch die Relation R.

X Aufruf eines Identifikators, der deklariert sein muss. Wechselseitige Rekursion ist dabei erlaubt.

Die Bedeutung von CSP kann durch eine Strukturierte Operationelle Semantik (SOS) gegeben werden. Hierbei werden die Prozesse als Zustände angesehen und die Ereignisse als Beschriftung von Transitionen. Des weiteren gibt es noch die Trace-Semantik, auch für CSP-OZ-DC verwendet wird, und die Failure-Divergences-Semantik. Traces sind endliche Sequenzen von sichtbaren Ereignissen. Um auch den Unterschied von deterministischer und nichtdeterministischer Auswahl zu erkennen, werden in der Failure-Divergences-Semantik zusätzlich zur Trace noch die Fehlermengen und Divergenzpunkt-Mengen verwendet. Divergenzpunkte sind Traces, nach denen eine unendliche Folge von τ-Transitionen (interne Aktionen) möglich ist.

2.1.2. OZ

Object-Z [Smi00] ist eine objektorientierte Erweiterung von Z [Spi92]. Die Sprache Z ist eine mathematische Notation, die an der Universität Oxford Anfang der 1980er Jahre entwickelt wurde [ASM80]. Sie besteht einerseits aus dem mathematischen Toolkit, das Standard-Notationen für logische Operationen, Funktionen, Relationen und Operationen auf Mengen bereitstellt. Andererseits wurde ein Schemakalkül entwickelt, bei dem Variablen in Schemata gekapselt werden und Operation auf diesen Schemas definiert werden, die diese inneren Variablen verändern. Die Daten werden entweder durch ein Schema der Form $Name \mathrel{\widehat=} [Deklaration \mid Praedikat]$ oder als Z-Schema-Box wie in Abbildung 1 dargestellt.

Um die Wirkung von Operationen darzustellen werden zusätzliche, dekorierte Versionen des Zustandsraumes eingeführt (Abbildung 2). Eine dekorierte Variable *x*’ beschreibt dabei die Variable *x* nach der Änderung.

Die Operationen selbst werden auch als Z-Schema dargestellt und enthalten die Daten mit und ohne Dekoration. Dies soll den Zustandsraum vor und nach der Operation beschreiben. In dem Beispiel (Abb. 3) ist *neu*? ein Eingabeparameter.

Fahrstuhl
$aktuellerStock : \mathbb{N}$
$aktuellerStock \geq 0$ $aktuellerStock \leq 3$

Abbildung 1: Z-Schema *Fahrstuhl*

Fahrstuhl
$aktuellerStock' : \mathbb{N}$
$aktuellerStock' \geq 0$ $aktuellerStock' \leq 3$

Abbildung 2: Dekoriertes Z-Schema *Fahrstuhl*

Mithilfe des Δ-Operators lassen sich die Z-Schemata für den undekorierten und dekorierten Zustandsraum zusammenfassen und das Schema für die Operation kann somit verkürzt werden (Abb. 4).

Die Grundeinheit einer Z-Spezifikation sind Paragraphen. Dies sind gegebene Typen, axiomatische Beschreibungen und die Schema-Definitionen. Es lassen sich auch eigene Typen mittels Freie-Typen-Paragraphen erstellen. In Z lässt sich der Typ jeder Variablen und jedes Ausdrucks statisch bestimmen. Die Semantik für Z wird im ISO/IEC-Standard für Z in einer Meta-Sprache definiert, die auf Zermelo-Fraenkel-Mengenlehre beruht.

2.1.3. DC

Der Duration Calculus [ZHR91] ist eine Realzeit-Intervall-Logik die mit einem Kalkül verknüpft ist. DC wurde ursprünglich im Zusammenhang mit dem ProCoS-Projekt (Provably Correct Systems) [JHF+94] entwickelt. Die Syntax von DC wird zunächst von folgender Menge von Symbolen gebildet:

SVar Menge der boolschen Zustand-Variablen (*Observablen*) $P, Q, R, ...$, deren Wert von dem jeweiligen Zeitpunkt abhängt

GVar Menge der Variablen $x, y, z, ...$ mit zeitunabhängigen $\mathbb{R}$-Werten

FSymb Menge der globalen Funktionen $f^n, g^m, ...$ mit Stelligkeiten $n, m \geq 0$. f^n erhält n Argumente vom Typ $\mathbb{R}$ und liefert einen Wert vom Typ $\mathbb{R}$

RSymb Menge der globalen Relationen $p^n, q^m, ...$ mit Stelligkeiten $n, m \geq 0$. p^n erhält n Argumente vom Typ $\mathbb{R}$ und liefert einen Wert vom Typ $\mathbb{B}$

$$
\begin{array}{|l}
\hline
\textit{fahreZu} \\
\textit{aktuellerStock} : \mathbb{N} \\
\textit{aktuellerStock}' : \mathbb{N} \\
\textit{neu}? : 0..3 \\
\hline
\textit{aktuellerStock} \geq 0 \\
\textit{aktuellerStock} \leq 3 \\
\textit{aktuellerStock}' \geq 0 \\
\textit{aktuellerStock}' \leq 3 \\
\textit{aktuellerStock}' = \textit{neu}? \\
\hline
\end{array}
$$

Abbildung 3: Z-Schema der Operation *fahreZu*

$$
\begin{array}{|l}
\hline
\textit{fahreZu} \\
\Delta \textit{Fahrstuhl} \\
\textit{neu}? : 0..3 \\
\hline
\textit{aktuellerStock}' = \textit{neu}? \\
\hline
\end{array}
$$

Abbildung 4: Z-Schema der Operation *fahreZu* mit Δ-Operator

Aufbauend auf diese Symbole besteht die Syntax von DC aus drei Teilen.

Eine Zustandszusicherung (oder auch Zustandsausdruck) π hat folgenden Aufbau:

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

Hierbei ist X eine Observable. Observablen werden verwendet um Systemeingaben und Systemausgaben zu modellieren. Die Semantik von Zustandszusicherungen ist gegeben durch eine Funktion $\mathcal{I}$. Sie ordnet einer Zustandszusicherung zu jedem Zeitpunkt einen boolschen Wert (0 für falsch oder 1 für wahr) zu.

$$
\begin{aligned}
\mathcal{I}[\![0]\!](t) &= 0 \\
\mathcal{I}[\![1]\!](t) &= 1 \\
\mathcal{I}[\![X = d]\!](t) &= \begin{cases} 1, & \text{gdw } X = d \\ 0, & \text{sonst} \end{cases} \\
\mathcal{I}[\![\neg\pi]\!](t) &= \begin{cases} 1, & \text{gdw } \mathcal{I}[\![\pi]\!](t) = 0 \\ 0, & \text{sonst} \end{cases} \\
\mathcal{I}[\![\pi_1 \wedge \pi_2]\!](t) &= \begin{cases} 1, & \text{gdw } \mathcal{I}[\![\pi_1]\!](t) = 1 \text{ und } \mathcal{I}[\![\pi_2]\!](t) = 1 \\ 0, & \text{sonst} \end{cases}
\end{aligned}
$$

Es lässt sich dann ein Term θ bilden mit folgendem Aufbau:

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

Hierbei ist x eine Variable, ℓ ein spezielles Symbol für die Intervalllänge, π eine Zustandszusicherung und f ein Funktionssysymbol. Die Semantik von Termen wird über ebenfalls durch eine Funktion $\mathcal{I}$ geliefert, die jedem Term, gegeben eine Belegung der Variablen $\mathcal{V}$ und ein Zeitinterval $[b, e]$, eine reelle Zahl zuordnet.

$$\begin{aligned}
\mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) &= \mathcal{V}(x) \\
\mathcal{I}[\![\ell]\!](\mathcal{V}, [b, e]) &= e - b \\
\mathcal{I}[\![\textstyle\int \pi]\!](\mathcal{V}, [b, e]) &= \int_b^e \pi_{\mathcal{I}}(t)dt \\
\mathcal{I}[\![f(\theta_1, \cdots, \theta_n)]\!](\mathcal{V}, [b, e]) &= \hat{f}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e]))
\end{aligned}$$

$\pi_{\mathcal{I}}$ ist die Interpretation von π, also eine totale Funktion $\mathbb{R} \to \{0, 1\}$, und $\hat{f}$ eine Interpretation von f, also eine totale Funktion $\mathbb{R}^n \to \mathbb{R}$. Schließlich können Formeln nach folgender Vorschrift gebildet werden:

$$F ::= p(\theta_1, \cdots, \theta_n) \mid \neg F \mid F_1 \wedge F_2 \mid \forall x : F \mid F_1;\ F_2$$

Hierbei ist p ein Prädikatssymbol, θ_1 bis θ_n Terme und x eine Variable. Die Semantik des DCs wird über eine Interpretationsfunktion gebildet. Formeln wird über eine Funktion $\mathcal{I}$, gegeben eine Belegung der Variablen $\mathcal{V}$ und einem Zeitintervall $[b, e]$, einen boolschen Wert zugeordnet.

$$\begin{aligned}
\mathcal{I}[\![p(\theta_1, \cdots, \theta_n)]\!](\mathcal{V}, [b, e]) &= \hat{p}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \\
&\quad \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e])) \\
\mathcal{I}[\![\neg F]\!](\mathcal{V}, [b, e]) &= \text{tt gdw } \mathcal{I}[\![F]\!](\mathcal{V}, [b, e]) = \text{ff} \\
\mathcal{I}[\![F_1 \wedge F_2]\!](\mathcal{V}, [b, e]) &= \text{tt gdw } \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, e]) = \text{tt und} \\
&\quad \mathcal{I}[\![F_2]\!](\mathcal{V}, [b, e]) = \text{tt} \\
\mathcal{I}[\![\forall x : F]\!](\mathcal{V}, [b, e]) &= \text{tt gdw für alle } d \in \mathbb{R} gilt : \\
\mathcal{I}[\![F]\!](\mathcal{V}[x := d], [b, e]) &= \text{tt} \\
\mathcal{I}[\![F_1;\ F_2]\!](\mathcal{V}, [b, e]) &= \text{tt gdw es ein } m \in [b, e] \text{ gibt, so dass} \\
&\quad \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, m]) = \text{tt und} \\
&\quad \mathcal{I}[\![F_2]\!](\mathcal{V}, [m, e]) = \text{tt}
\end{aligned}$$

$\hat{p}$ ist die Interpretation von p, also eine totale Funktion $\mathbb{R}^n \to \mathbb{B}$, und $\mathcal{V}[x := d]$ bedeutet, dass in der Variablenbelegung $\mathcal{V}$ an die Variable x einen neuen Wert d erhält. Zur Vereinfachung der Formeln können folgende Abkürzungen verwendet werden:

$$\begin{aligned}
\text{Punktintervall} &: \lceil\rceil := \ell = 0 \\
\text{Überall} &: \lceil\pi\rceil := \ell > 0 \wedge \textstyle\int \pi = \ell \\
\text{Irgendwann} &: \Diamond F := true;\ F;\ true \\
\text{Immer} &: \Box F := \neg\Diamond\neg F
\end{aligned}$$

2.1.4. CSP-OZ-DC-Klassen

Die drei zuvor genannten Spezifikationen behandeln jeweils verschiedene Aspekte eines Systems (Daten, Prozess-Kommunikation, zeitliches Verhalten). Um diese Aspekte gleichzeitig in einer Sprache zu behandeln, wurden die Sprachen zusammengefasst. C.Fischer kombinierte CSP und OZ und entwickelte die integrierte Spezifikationssprache CSP-OZ [Fis00, Fis97]. Objektorientierten Konzepte, wie z.B. Erzeugung mehrerer Instanzen einer CSP-OZ-Klasse und Vererbung, machen CSP-OZ zu einer objektorientierten Spezifikationssprache. Zugriff auf andere Objekte mittels Referenzen findet auf CSP-Ebene statt. Schließlich wurde der DC-Teil hinzugefügt [HO02a].

Diese Kombination aus den drei Spezifikationssprachen hat eine kompositionelle Semantik als Parallelkombination von Phasen-Event-Automaten, die sich sehr gut als Zwischendarstellung bei der Übersetzung von CSP-OZ-DC in die Eingabesprache eines Abstraction Refinement Model Checkers (ARMC) eignen. Mit ARMC lässt sich die Erreichbarkeit von Zustandsmengen automatisch überprüfen.

2.2. UML-Profil für CSP-OZ-DC

Die UML hat keine exakte Semantik, und mit ihr spezifizierte Systeme können somit nicht automatisch verifiziert werden. Die Verifikation kann aber auf CSP-OZ-DC-Ebene automatisch durchgeführt werden, was bedeutet, dass zuerst eine Transformation in diese Ebene erfolgen muss. Möglich ist diese Umwandlung durch UML-Profile, bei denen nur eine Untermenge der möglichen UML-Diagrammarten verwendet wird. Ein UML-Profil ist ein Modell-Element in der UML, das eine Erweiterung des UML-Metamodells mithilfe von *Stereotype* und *Tags* erlaubt. *Stereotype* sind spezielle UML-Klassen und werden verwendet, um Modell-Elemente aus dem UML-Metamodell zu erweitern, um z.B. einen Verwendungskontext aus der modellierten Domäne widerzuspiegeln. *Tags* definieren neue Attribute in den *Stereotypen*.

Das entworfene UML-Profil enthält die UML-Elemente Klassendiagramm und Statemachines. In dem UML-Profil UML-RT [FOW01, SR98], welches ebenfalls Aspekte von Realzeit behandelt, werden sogenannte Strukturdiagramme definiert. Die Diagrammart aus UML 2.0, die diesen Strukturdiagrammen entspricht, ist die der Komponentendiagramme, welche ebenfalls zum neuen Profil hinzugefügt werden.

Das UML-Profil liefert *Stereotype* und *Tag*s für die CSP-OZ-DC-Klassen und ihrer Inhalte, sowie die Stereotype *capsules*, *protocols* und *ports* aus UML-RT, bzw. aus den UML-Komponentendiagrammen. *Capsules* sind Objekte, die in einem eigenen Thread laufen, und die eigentlichen Systembausteine. Sie benutzen Operationen, oder stellen welche zur Verfügung, wobei der Zugang zu den Operationen durch Sender-*Ports*(*base*, schwarz) und durch Empfänger-*Ports*(*conjugated*, weiß) definiert ist. Weder Attribute noch Methoden einer *Capsule* sind nach außen hin sichtbar. *Protocol* regelt die Kommunikation zwischen zwei oder mehreren Ports und bestimmt, auf welche Operationen zugegriffen werden darf. Dabei beschreibt es mithilfe der Stereotype *in* und *out* (bzw. der Broadcasting-Versionen *in_S* und *out_S*) die Rolle der Methoden. Das Profil definiert Ports als eine Assoziation zwischen Capsules und Protocols mit einer Aggregation auf Seiten der Capsule. Capsules und Protocols enthalten folgende Tags:

Klasse vom Typ *Capsule*
invariant
die Klasseninvariante als Z-Prädikat
init
Initialisierungszustand der Klasse als Z-Prädikat
dc counterexample
DC-Gegenbeispielformeln die für die Klasse gelten

Operationen der Klasse(alles Z-Prädikate)
changes
Attribute, die durch Operation verändert werden
enable
Vorbedingungen für die Ausführung der Operation
effect
Änderung der Attribute bei Ausführung der Operation

Operationen einer Klasse vom Typ *Protocol*(alles Z-Prädikate)
input
Eingabeparameter
output
Ausgabeparameter
simple
Adressinformationen zur Adressierung einer Instanz

Einschränkungen:

- Eine erbende Capsule muss die Statemachine der Ober-Capsule implementieren und kann höchstens eine neue Statemachine parallel dazu ausführen.
- *base*-Ports können nur mit *conjugated*-Ports verbunden werden. Ausnahme ist die Delegation von Portfunktionen von äußeren Capsules auf innere.

Die folgenden Regeln zur Übersetzung der UML in die Systembeschreibungssprache CSP-OZ-DC geben den Diagrammen eine eindeutige Semantik, die auch eine automatische Verifikation der entwickelten Systeme ermöglicht. Klassendiagramme, Statemachines und Strukturdiagramme werden getrennt übersetzt. Die einzelnen Punkte der Übersetzung werden hier nur kurz aufgeführt. Für eine ausführliche Beschreibung wird auf [FOW01] und [RW03] verwiesen, ein komplexes Beispiel liefert u.a. [MORW08, MORW04].

Übersetzungsregeln für die UML-Elemente

Klassendiagramme

- normale Datenklassen in OZ-Klassen umwandeln
- capsules in CSP-OZ-DC-Klassen umwandeln
- Attribute aus Diagramm übernehmen
- Datenklassen-Attribute in Visibility-Liste eintragen
- assoziierte Datenklassen einer Capsule als Attribut eintragen
- tags *init*, *invariant* und *dc counterexample* erhalten entsprechende Blöcke in der CSP-OZ-DC-Klasse
- Operationen des *Protocol*s in das Interface der assoziierten Klasse eintragen; Rolle des Ports bestimmt Art der eingetragenen Operation: bei *base*-Ports wird *in*/*in_S* zur *method*-Operation, *out*/*out_S* zur *chan*-Operation, bei *conjugated*-Ports werden vor der Umwandlung die Rollen invertiert
- tags der Operationen mit übernehmen

Statemachines

- Übersetzungsabbildung $\varphi : SM \rightarrow CSP$ auf die gesamte Statemachine und all ihre Untermaschinen anwenden [RW03]:

$$\varphi \equiv \begin{cases} P_s = SKIP & \text{wenn } s \text{ Endzustand ist} \\ P_s = \Box_{(e,t)\in\mathcal{T}_s}\, e \rightarrow P_t & \text{wenn } s \text{ einfacher Zustand ist und } C_s = \varnothing \\ P_s = \sqcap_{t\in C_s} P_t & \text{wenn } s \text{ einfacher Zustand ist und } C_s \neq \varnothing \\ & \text{oder wenn } s \text{ Initialzustand ist} \\ P_s = (|||_{i=1}^{n} P_{M_i}); & \\ \textbf{if}\, C_s \neq \varnothing\, \textbf{then}\, (\sqcap_{t\in C_s} P_t)\, \textbf{else}\, STOP & \text{wenn } s \text{ zusammengesetzter Zustand ist} \\ & \text{mit Untermaschinen } M_i,\, 1 \leq i \leq n. \end{cases}$$

$\varphi(M) \equiv P_M = \sqcap_{t\in\mathcal{I}_M} P_t$ für jede Untermaschine M von SM.

Hierbei gelten folgende Definitionen:

SM bezeichnet eine Statemachine

s Zustand der Statemachine

CSP Menge aller CSP-Prozesse

P_s,P_t CSP-Prozesse

$\mathcal{C}_s$ Menge aller direkten Nachfolger von s ohne Triggerevent

$\mathcal{T}_s$ Menge aller Paare (e,t) der direkten Nachfolger t von s mit Triggerevent e

$\mathcal{I}_M$ Menge der Initalzustände einer Untermaschine M von SM

- das Ergebnis als CSP-Spezifikation in die zur Statemachine gehörige Klasse eintragen

Komponentendiagramme
Diese Übersetzung ist nur zur Vollständigkeit angegeben. Bisher ist es noch nicht möglich, mehrere Instanzen einer Capsule zu übersetzen, sodass dies auch noch nicht in Syspect implementiert ist.

- für jede Instanz einer Capsule wird ein (CSP-)Prozess parallel erzeugt
- Prozesse zweier über *base/conjugated*-Ports verbundener Capsules laufen synchronisiert auf den definierten Operationen des zugehörigen Protocols ab
- bei Broadcast-Operationen müssen alle daran beteiligten Instanzen auf ihnen synchronisiert werden
- alle internen Operationen werden als *local_chan* definiert
- Operationen, die über Ports nach Außen durchgeschleift werden, werden in die CSP-OZ-DC-Klasse eingetragen

2.3. Phase-Event-Automaten als Semantik von CSP-OZ-DC

Es lässt sich somit mittels eines UML-Profils ein System mit zeitlichen Eigenschaften beschreiben, wobei die Diagramme eine eindeutige Semantik in CSP-OZ-DC haben. In [HM05] wird eine neue Automatenklasse, die der Phasen-Event-Automaten eingeführt. PEAs bilden eine operationelle Semantik für alle drei Teile von CSP-OZ-DC, was Voraussetzung für automatisches Model-Checking ist [CGP99]. Hier sollen nun die einzelnen Übersetzungen der drei CSP-OZ-DC-Teile in PEAs kurz dargestellte werden. Als Ergebnis erhält man das Produkt mehrerer paralleler Automaten, von denen jeder mit einem der CSP-OZ-DC-Teile korrespondiert.

2.3.1. Phasen-Event-Automaten

Ein PEA ist ein 8-Tupel (P,V,A,C,E,s,I,P_0). Die einzelnen Komponenten sind:

- P: Zustandsmenge (Phasen)
- V: Menge von Zustandsvariablen
- A: Menge von Events
- C: Menge der Uhren

- $E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathcal{P}(C) \times P$: Transitionsrelation
- $s : P \to \mathcal{L}(V)$: Invarianten-Prädikat für Zustände
- $I : P \to \mathcal{L}(C)$: Uhreninvariante für Zustände
- $P_0 \subseteq P$: Startzustände.

Alle Uhren werden anfangs auf Null gesetzt und messen im weiteren Verlauf die Zeit, die seit dem letzten Rücksetzen vergangen ist. Ein PEA startet in einem seiner Startzustände, dessen Invarianten erfüllt sind. Jede Variablenveränderung und jedes Auftreten eines Events sorgt für einen Transitionsschritt. Ein Element (p, g, X, p') aus der Menge der Transitionen repräsentiert eine Kante von der Phase p zur Phase p', wobei der *Guard* g für die aktuelle Uhren-Belegung, die Vor- und Nachbelegung der Zustandsvariablen und die auftretenden Events erfüllt sein müssen. Außerdem kann ein Phasenwechsel nur stattfinden, wenn die Uhren- und Zustandsvariablen die Invarianten für die neue Phase erfüllen[1]. Alle Uhren aus der Menge X werden zurückgesetzt. Damit Automaten auch Schritte durchführen kann, wenn kein für sie relevantes Event stattfindet, bzw. die Zustandsvariante der aktuellen Phase sich nicht ändert, erhalten alle Phasen sogenannte Stotternde Kanten (stuttering edges [Lam83]). Diese Schleifen sind wichtig für die spätere parallele Komposition mehrerer PEAs, da erst durch sie synchrone Schritte möglich sind.

2.3.2. Übersetzung von CSP-OZ-DC in Phasen-Event-Automaten

CSP
Für die Übersetzung wird die operationelle Semantik genutzt. Die Phasen werden mit den CSP-Prozessen beschriftet, als Alphabet (Menge der *events*) wird das Alphabet von `main` übernommen. Variablen und Uhren sind in diesem PEA nicht vorhanden. Für jede Transition $p \xrightarrow{a} p'$ der operationellen Semantik gibt es ein Kante:

- $(p, a \wedge \bigwedge_{e \in A \setminus \{a\}} \neg e, \varnothing, p') \in E$ falls $a \neq \tau$
- $(p, \bigwedge_{e \in A} \neg e, \varnothing, p') \in E$ sonst

Zusätzlich erhält jeder Zustand eine sogenannte *Stotternde Kante* (stuttering edge):

- $(p, \bigwedge_{e \in A} \neg e, \varnothing, p) \in E.$

Benötigt wird diese später bei der parallelen Komposition. Die Startzustände werden vom `main`-Prozess abgeleitet.

Object-Z
Der Object-Z-Teil wird in einen 2-Phasen-Automaten mit den Phasen *init* und *main* übersetzt, im Gegensatz zu [Hoe06], bei der es statt der init-Phase eine initiale Kante gibt. Die

[1] In jedem Zustand muss Zeit vergehen und somit muss auch nach einer Transition die Uhreninvariante noch für kurze Zeit gelten. [Hoe06] definiert dazu den *strict*-Operator.

`main`-Phase hat für jedes Event eine Kante, die genau dieses Event erlaubt, die Variablen, die nicht in der Δ-Liste stehen, konstant hält und die Variablen in der Δ-Liste entsprechend dem Kommmunikations-Schema beschränkt. Weiterhin hat jede Phase wieder die *Stotternde Kante*, die alle Ereignisse und Variablenänderungen verbietet.

Duration Calculus
Eine Teilsprache des Duration Calculus, die Menge der DC-Gegenbeispiel-Formeln (dc counterexample formulae), lässt sich in einen Phase-Event-Automaten übersetzen. DC-Gegenbeispiel-Formeln haben die Struktur:

$$\neg(phase_1 \,;\, \cdots \,;\, phase_i)$$

und verbieten somit den Trace $(phase_1 \,;\, \cdots \,;\, phase_i)$. Der Übersetzungsvorgang entspricht im Kern dem Algorithmus zur Umwandlung von nichtdeterministischen in deterministische endliche Automaten (NEA → DEA): Es werden Potenzmengen aus den einzelnen DC-Phasen konstruiert. Dazu wird, beginnend mit der ersten DC-Phase, die DC-Gegenbeispiel-Formel durchlaufen und es soll sich für jede dieser Phasen gemerkt werden, ob das Zeitintervall vom System-Start bis zum jetzigen Zeitpunkt die Formel

$$\text{true} \,; phase_1 \,;\, \cdots \,;\, phase_i, \qquad 1 \leq i \leq n$$

erfüllt. Eine Phase im PEA wird jeweils mit einer Menge solcher DC-Phasen aus der Gegenbeispiel-Formel beschriftet, in denen sich die DC-Phase zum jeweiligen Zeitpunkt befinden kann. Enthält ein DC-Phase eine Beschränkung der Dauer, so muss die PEA-Phase über eine Uhr verfügen, die im Verlauf folgendermaßen zurückgesetzt wird:

- so oft wie möglich, falls sie eine obere Schranke misst
- nur beim Neubetreten der Phase, falls sie eine untere Schranke misst

PEA-Phasen, die den Zustand widerspiegeln, in dem der ganze, durch das Gegenbeispiel spezifierte Trace durchlaufen wurde, werden nicht mit in den PEA aufgenommen, um dieses Verhalten zu verbieten.

2.4. Eclipse

Laut der Eclipse-Homepage ist Eclipse „ein Framework für alles Mögliche“. Hauptsächlich wird es aber als IDE[2] für Java verwendet. 2001 wurde die erste Eclipse-Version veröffentlicht und hat bis heute eine große Beliebtheit erlangt. Grund dafür ist sicherlich, dass namhafte Unternehmen wie IBM, Borland, SAP, Nokia und Intel an der Entwicklung beteiligt sind, auch die Tatsache, dass Eclipse Open-Source ist, trägt zum Erfolg bei. Das Framework verwendet Konzepte von OSGi[3] [WHKL08] zur dynamischen Modularisierung, Versionierung und Wiederverwendung und besteht, bis auf das Laufzeitmodul, aus Plugins. Ein wichtiger und aktueller Aspekt von Eclipse (seit Version 3.0) ist die Möglichkeit, Rich-Client-Applications auf Basis von Rich-Client-Platforms (RCP) zu entwickeln.

[2]Integrated Development Environment
[3]Open Services Gateway initiative

2.4.1. Plugins

Plugins entsprechen den Bundles der OSGi-Spezifikation. Programme lassen sich somit aus einer beliebigen Anzahl (auch schon vorhandener) Plugins zusammensetzen. Eclipse ermöglicht die Entwicklung eigener Plugins mit dem Plugin-Development-Environment (PDE). Plugins können auf andere Plugins aufbauen, indem sie deren Extension-Points erweitern. Diese Erweiterungspunkte und Erweiterungen werden in der Manifest-Datei *plugin.xml* deklariert. Die implementierende Klasse wird erst zu Laufzeit geladen (Lazy Loading), was die Performance des Programms stark verbessert.

2.4.2. RCP - Rich Client Platform

Als RCP wird die minimale Anzahl an Plugins definiert, die für die Ausführung einer Rich-Client-Application außerhalb der Eclipse-IDE notwendig ist. Rich-Clients enthalten applikationsspezifischen Code, im Gegensatz zu den Thin-Clients, deren Funktionalität vom Server aus gesteuert wird. Typischerweise enthalten Eclipse-RCPs Viewer, Editoren, Perspektiven und Menüs. Nachdem man ein Programm durch Auswahl von Plugins, Erweiterungen usw. entwickelt hat, kann man daraus dann ein auslieferungsfähiges Produkt machen, indem man den Erweiterungspunkt *org.eclipse.core.runtime.products* erweitert oder die *config.ini* anpasst, um z.B. Willkommen- oder About-Seiten hinzuzufügen.

2.5. GEF - Graphical Editing Framework

Das Eclipse-Plugin GEF [Dau07] ist ein Framework zur Darstellung für Modelle im Sinne des Model-View-Controller-Musters (MVC, [Ree79b], [Ree79a], [LR09]). Es erlaubt dem Benutzer, mit dem Modell zu interagieren. Benutzereingaben durch Maus oder Tastatur können bearbeitet und interpretiert werden, das Modell kann verändert werden und Änderungen lassen sich wieder rückgängig machen (undo/redo). Nützliche Workbench-Funktionen wie Aktionen und Menüs, Toolbars und Key-Bindings erhöhen die Benutzerfreundlichkeit mit GEF erstellter Programme.
In der *Model*-Schicht befinden sich die darzustellenden persistenten und wichtigen Daten. Das Modell kennt die anderen Schichten nicht und teilt Änderungen über Listener mit. Im Gegensatz dazu enthält die *View*-Ebene keine Daten und keine Modell-Logik und kennt die anderen Schichten nicht. Das Verbindungsglied der *Model*- und der *View*-Schicht bildet der *Controller*. Kommunikation vom Modell wird an die *View*-Schicht weitergeleitet. In GEF sind die Controller Unterklassen von EditPart, wobei jedem EditPart genau ein Modell und eine View zugeordnet ist. Die Darstellung auf der View-Ebene erfolgt durch Figures aus dem Eclipse-Plugin Draw2D[4], das u.a. auch Layout-Manager zur automatischen Anordnung von Figures zur Verfügung stellt. Figures werden in einer Baumstruktur verwaltet und zeichnen sich und rekursiv ihre Kinder selbst. Dem entsprechend in einer Baumstruktur aufgebaut sind auch die EditParts. Deren wichtigsten Methoden ermöglichen die Erstellung ihrer zugehörigen Figure, die Aktualisierung der Daten der View-Schicht mit den Daten der Modell-Schicht und die Rückgabe der logischen Kinder von dem zum EditPart korrespondierenden Objekt aus

[4]http://help.eclipse.org/ganymede/topic/org.eclipse.draw2d.doc.isv/guide/guide.html

der Modell-Schicht. Die Erzeugung des EditParts erledigt die EditPartFactory, nachdem ihr ein neu erstelltes Modell-Objekt übergeben wurde. Ein neues EditPart registriert sich dann beim Modell-Objekt, um über Änderungen informiert zu werden. Für die Behandlung der Benutzeraktionen sind die EditPolicies und die Commands zuständig. EditPolicies legen die Aufgaben oder Rollen eines EditParts fest (z.B. Layout, Connection, GraphicalNode). Führt der Benutzer eine Aktion durch, wie z.B. das Verschieben eines Knotens, so werden zunächst eine Reihe von Events vom Standard-Widget-Tool (SWT), das eine Schicht unterhalb von Draw2D angesiedelt ist, ausgelöst. Das System erstellt hierzu dann Requests, die über die EditParts an die EditPolicies geleitet werden. Aus den Requests erzeugen die EditPolicies dann Commands, die wiederum über die EditParts und das System auf dem CommandStack abgelegt werden. Durch die Commands wird die Undo/Redo-Funktionalität realisiert, da sie die Informationen zur Modell-Änderung und zum Verwerfen dieser Änderung enthalten.

2.6. SWT und JFace

Das Standard-Widget-Toolkit wurde speziell für Eclipse als Bibliothek für die Erstellung grafischer Oberflächen mit Java entwickelt. Im Gegensatz zu Swing verwendet SWT die nativen grafischen Elemente des Betriebssystems. Die Widgets (Steuerelemente) werden mithilfe dünner Wrapper eingebunden. Während SWT die Basis-Widgets liefert, setzt JFace diese zu komplexeren Widgets, wie z.B. Wizards und Dialoge zusammen. JFace erweitert außerdem SWT um das MVC-Konzept und enthält Viewer-Klassen, um Daten und ihre Darstellung miteinander zu verknüpfen, Actions-Klassen, um benutzerdefinierte Menü- oder Steuerknopf-Aktionen zu ermöglichen und Registraturen (Registries) für Schriftarten und Bilder.

2.7. Syspect

Das System Specification Tool, kurz Syspect, ist eine Modellierungsumgebung die aus einer Vielzahl von Plugins besteht, u.a. für verschiedene Diagramm-Editoren, Exporte, Persistenz und Konsistenz. Die sogenannten aktiven Plugins erweitern Extension-Points. Diese werden von Eclipse und auch von Syspect selbst zur Verfügung gestellt, um die Integration des jeweiligen Plugins in das Programm zu ermöglichen.

2.7.1. PEA-Toolkit

Auch das PEA-Toolkit wird in Form eines Plugins in Syspect eingebunden. Es wurde von Jochen Hoenicke, Roland Meyer und Johannes Faber entwickelt und bietet neben der Möglichkeit der Repräsentation von PEAs in Java u.a. noch die Konvertierung von PEA nach XML, die Umwandlung von DC-Countertrace-Formeln in PEA, die Berechnung des Parallelprodukts von PEAs und einige Exporte, wie z.B. in das Eingabeformat für ARMC.

2.7.2. Community-Z-Tools

Das Projekt Community-Z-Tools (CZT) stellt mit seiner Werkzeugsammlung für die Spezifikationssprache Z, und teilweise auch Object-Z, mehrere Funktionalitäten für Syspect zur

Verfügung. So liefert es u.a. die Speichermöglichkeit im Format ZML, einen Parser und einen Type-Checker für Z-Ausdrücke.

3. Anforderungen

3.1. Bestandsanalyse: PEAs in Syspect

Zur Darstellung der funktionellen Anforderungen soll zunächst einmal gezeigt werden, welche Rolle die Phasen-Event-Automaten in der momentanen Version von Syspect einnehmen. PEAs sind in Syspect nur ein für den Benutzer verborgenes Zwischenprodukt beim Export von einer CSP-OZ-DC-Spezifikation. Lediglich beim expliziten PEA-XML-Export werden die PEAs sichtbar.

CSP-OZ-DC-Spezifikationen lassen sich sowohl ganz oder auch teilweise in PEAs als XML exportieren oder in die Eingabesprache für ARMC, die Transition-Constraint-Systeme, umwandeln. Ein Zwischenschritt in der Erzeugung der Exporte ist die Generierung von *PhaseEventAutomata* aus dem PEA-Toolkit. Das XML-Format für PEAs wird ausführlich in der Anforderungsdefinition zu Syspect beschrieben und beruht auf dem Format [Mey05] von Roland Meyer. Den ARMC-Export und die Verifikation einer CSP-OZ-DC-Spezifikation mittels ARMC beschreibt [Hob07].

3.1.1. MobyPEA

Einen externen Editor für PEAs stellt das Tool Moby/PEA zur Verfügung. Moby/PEA ist ein Produkt des MOBY-Projekts[5] der Abteilung *Entwicklung korrekter System* (kurz: CSD, Correct Systems Design Group) im Department für Informatik an der Universität Oldenburg. Leider ist Moby/PEA nur unter Linux lauffähig, während Syspect als Java-Projekt plattformunabhängig ist. Außerdem gibt es Inkonsistenzen zwischen der PEA-Version des PEA-Toolkits und der von Moby. Trotzdem kann Moby/PEA als eine gute Vorlage für den neuen internen PEA-Editor dienen, da es im Laufe der Zeit erweitert und den Bedürfnissen der Benutzer angepasst wurde.

3.2. Vorgaben an den Syspect PEAEditor

Der neu zu erstellende Editor für Phasen-Event-Automaten soll sowohl visuell als auch funktionell passend in das Spezifikations-Werkzeug Syspect integriert werden, d.h. der Benutzer von Syspect soll auch den PEA-Editor in gewohnter Art und Weise bedienen können. Dies bedeutet, dass die PEAs und ihre Kind-Elemente in der Navigator-View aufgelistet werden, dort selektierbar und über Menü oder Popup-Menü eine Aktion, wie z.B. exportieren oder löschen, auf ihnen ausgeführt werden kann. Ein Doppel-Klick auf ein PEA öffnet das entsprechende Diagramm im Editor. Dieser stellt Funktionen zum Erstellen, Verändern und Löschen von PEA-Elementen zur Verfügung. Das Aussehen der PEAs soll dem aus der Literatur entsprechen, bzw kann hier als Vorlage auch Moby/PEA dienen. Die Eigenschaften der im Editor oder im Navigator selektierten Elemente werden über die Properties-View geändert. Neben den reinen Editor-Funktionen sollen auch Import- und Exportfunktionen zur Verfügung gestellt werden. Dazu sollen entsprechende Aktionen für solche Elemente selektierbar sein, für die diese Funktionen sinnvoll sind. Die Begriffe Import und Exporte sind hier aus Sicht des PEA-Editors zu verstehen, auch wenn es sich eventuell nur um Syspect-interne Konvertierungen handelt.

[5] http://csd.informatik.uni-oldenburg.de/ moby/

Syspect besteht aus einer Menge von Plugins, und auch der PEA-Editor soll als Plugin entworfen werden. Damit Syspect auch ohne das PEAEditor-Plugin lauffähig bleibt, sind keine Abhängigkeiten der anderen Plugins von diesem erlaubt.

3.2.1. PEA-Projekte

Ein schwieriger Punkt ist die sinnvolle Einbettung der PEAs in Bezug auf die vorhandenen Syspect-Projekte. Es besteht auf der einen Seite die Möglichkeit, gemischte Projekte zu erstellen, indem PEAs zu den Syspect-Projekten hinzugefügt werden, entweder durch Konvertierung (z.B. aus eine Capsule) oder durch Erstellung neuer, andererseits kann man diese gemischten Projekte verbieten und somit nur reine PEA-Projekte zulassen.

Die PEAs ergänzen eine Spezifikation in Syspect durch eine andere Darstellungsform, d.h. wenn aus einer Capsule ein PEA erzeugt wird, ändert sich nichts an der Semantik der Spezifikation. Wenn neue PEAs zu einem Projekt hinzugefügt werden kann sich das Verhalten aber ändern, denn die Semantik der CSP-OZ-DC-Spezifikation ist die Parallel-Komposition mehrerer PEAs, und neue PEAs werden dieser Komposition hinzugefügt.

Problematisch sind vor allem Änderungen nach dem Konvertieren von PEAs aus der CSP-OZ-DC-Spezifikation, seien es Änderungen an den PEAs oder an der Ursprungs-Spezifikation. Es ist kaum möglich, beides synchron zu halten, lediglich für Namen von Attributen, Variablen, Methoden oder Events könnten Modifikationen mit einigem Aufwand behandelt werden. Werden aber beispielsweise DC-Formeln in der Spezifikation geändert, müssen die PEAs neu erzeugt werden.

Eine Lösung bei diesen Misch-Projekten wäre es, das Editieren von den PEAs zu verbieten, sodass der Benutzer nur eine Ansicht der PEAs, also der Semantik der CSP-OZ-DC-Spezifikation, bekommt. Bei Änderungen an der Spezifikation müsste der Anwender die PEAs dann selber wieder neu erzeugen. Neue PEAs könnten jedoch hinzugefügt werden und auch in einen späteren Export, z.B. für den ARMC-Model-Checker, mit einbezogen werden. Hier gibt es aber auch wieder das Problem, dass der Benutzer seine Variablen-Namen mit Bedacht wählen muss, damit auch das gewünschte Verhalten von den neuen PEAs zusammen mit den beim Export erzeugten PEAs modelliert wird. Insgesamt ist der Ansatz mit gemischten Projekten aber für den Anwender zu fehleranfällig.

Die andere Möglichkeit ist es, nur reine PEA-Projekte zuzulassen. Der Anwender wählt eine CSP-OZ-DC-Spezifikation, bzw. die zugehörigen UML-Elemente, aus und konvertiert sie zu PEAs. Diese werden entweder in ein neues oder vorhandenes PEA-Projekt geschrieben. Jetzt kann der Benutzer sich die PEAs anzeigen lassen, um sie auf eventuelle Fehler in der Spezifikation zu untersuchen, oder die PEAs verändern, bzw. neue erstellen , um beispielsweise die Auswirkungen mit einem anschließenden Model-Check zu überprüfen. Der Bezug zur ursprünglichen CSP-OZ-DC-Spezifikation besteht durch die bei den konvertierten PEAs verwendeten Namen. Die PEAs erhalten den Namen der zugehörigen Capsule (bei CSP-Teilen als Ursprung mit dem Präfix „CSP“) oder der zugehörigen DC-Formel. Namen von Variablen oder Events werden komplett von den Attributen oder Methoden übernommen. Für den Benutzer bleiben die Projekte mit dieser Version übersichtlicher. Verschiedene Darstellungen mit derselben Semantik, d.h. Teile der CSP-OZ-DC-Spezifikation und zugehörige PEAs, treten hier nicht auf.

Da die zweite Variante mit reinen PEA-Projekten wesentlich sinnvoller erscheint, wird diese bevorzugt.

3.2.2. Sammlung von PEAs

Die PEAs des PEA-Toolkits treten bei den Exporten nicht einzeln in Erscheinung, sondern als Elemente eines Containers, dem PEANet. Das PEANet enthält neben den PEAs auch globale Deklarationen. Das Modell im PEAEditor soll ebenfalls Container für die PEAs anbieten. Neben den Deklarationen ist es aus Anwendersicht komfortabel, wenn für den Container zusätzlich die Variablen, Uhren und Events aus den PEAs zusammengefasst angezeigt werden.

3.3. Anwendungsbeispiel ”Elevator”

Als begleitendes Beispiel in dieser Arbeit wird das Fallbeispiel ”Elevator” aus [Hoe06] benutzt, das schon im vorigen Grundlagen-Abschnitt erwähnt wurde. Dieses Kapitel macht noch einmal deutlich, wie die Visualisierung und Bearbeitung von PEAs ohne den internen PEA-Editor abläuft.

„Elevator“ ist ein kleines Beispiel für eine CSP-OZ-DC-Spezifikation einer einfachen Fahrstuhl-Steuerung. Das Beispiel ist abstrakt gehalten, indem als unterstes und oberstes Stockwerk keine konkreten Werte angegeben sind, sondern nur die Attribute Min und Max. Wird ein neues Ziel von „außerhalb“ gewählt, setzt sich der Fahrstuhl in die gewünschte Richtung in Bewegung. Erreicht er die nächste Ebene, kann er entweder stoppen oder seine Fahrt fortsetzen. Zusätzlich werden durch DC-Formeln bestimmte zeitliche Anforderungen gestellt.
Es wird zunächst die Spezifikation des Fahrstuhls mittels Syspect modelliert. Die CSP-OZ-DC-Klasse wird durch eine *Capsule* und ein Interface in einem Klassendiagramm beschrieben. Die einzelnen OZ-Elemente werden in der `TabbedPropertyView` für die selektierte *Capsule* definiert (Tab ’Properties’ in Abbildung 5). Die zur *Capsule* gehörige *Statemachine* modelliert den CSP-Teil (Abb. 6),während die DC-Formeln der Capsule ebenfalls über die TabbedPropertyView hinzugefügt werden (Abb. 7).

Das Produkt des LaTeX-Exports ist in Abbildung 8 zu sehen. Dann werden aus dieser Spezifikation PEAs in Form einer XML-Datei erzeugt (Abb. 9).

Diese Datei wird dann anschließend in Moby/PEA (unter Linux) importiert, und die PEAs können dort angezeigt und bearbeitet werden (Abb. 10 und 11).

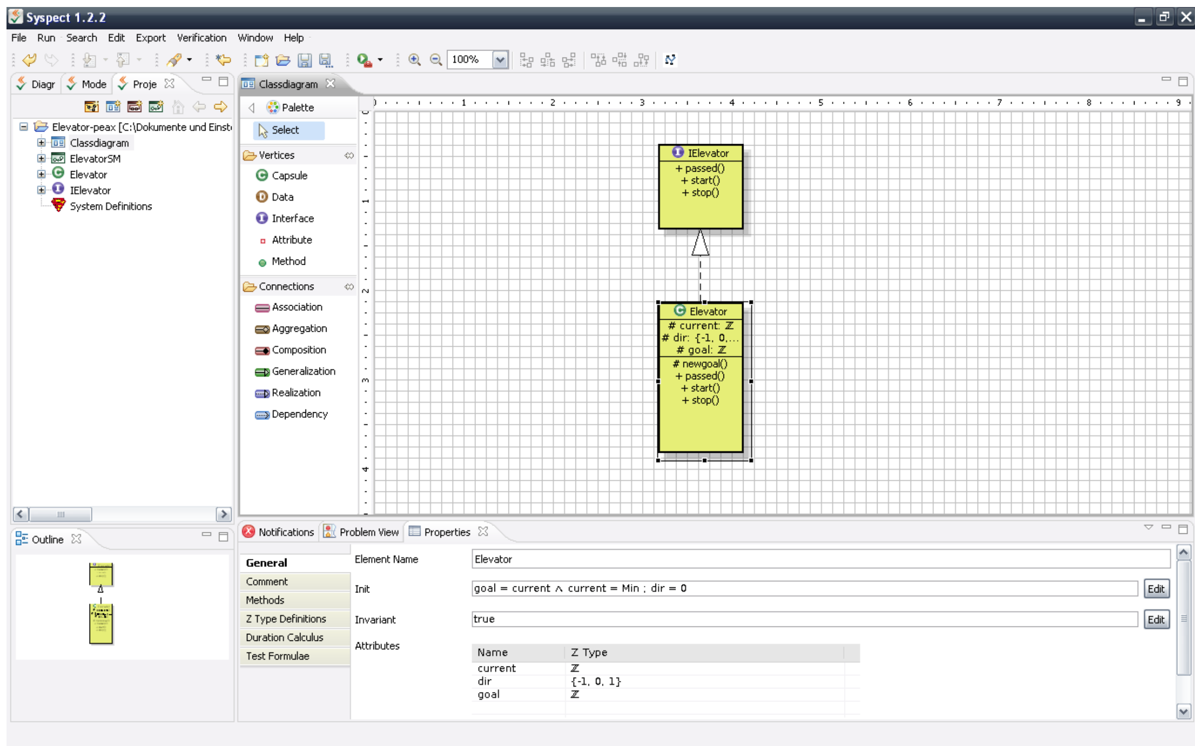

Abbildung 5: Syspect-Klassendiagramm mit TabbedPropertyView

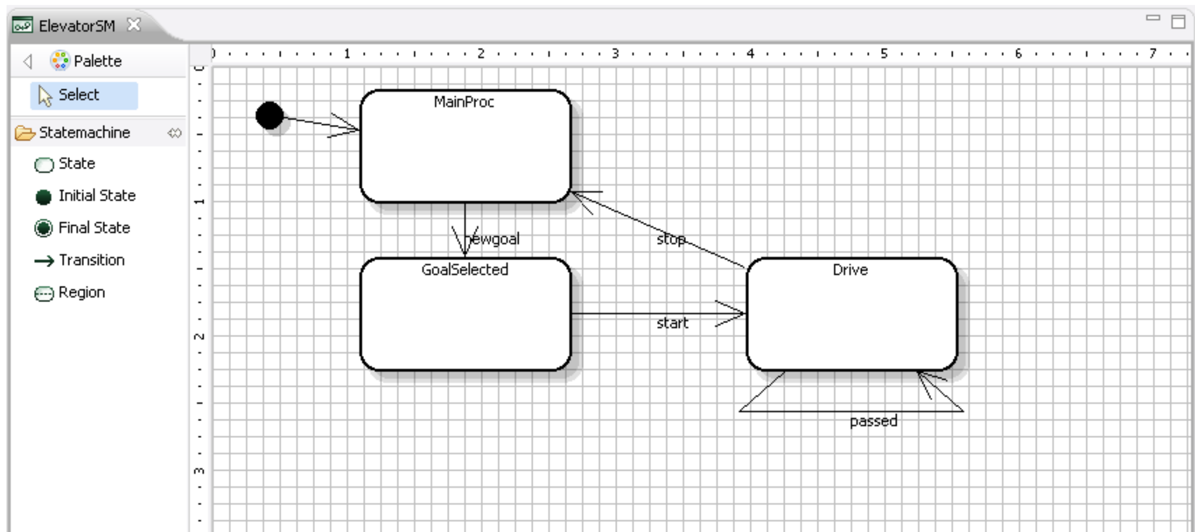

Abbildung 6: Syspect-Statemachine

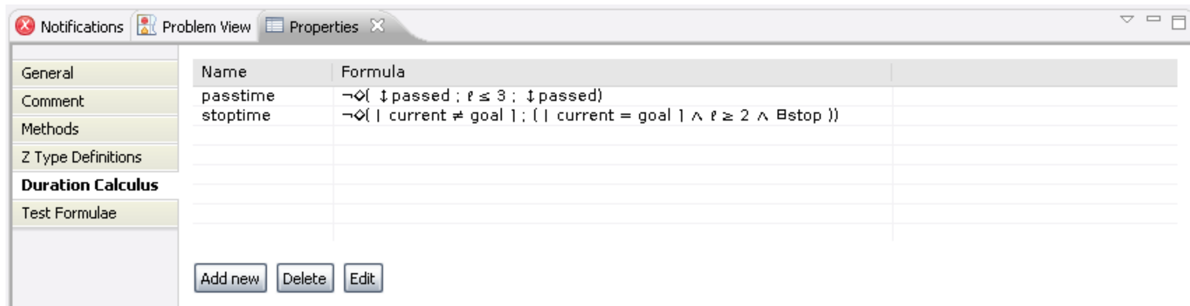

Abbildung 7: DC-Specifikation

Elevator

$\texttt{method}\ passed$
$\texttt{method}\ start$
$\texttt{method}\ stop$
$\texttt{local_chan}\ newgoal$

$Min, Max : \mathbb{Z}$

$Min < Max$

$$
\begin{aligned}
Drive &\overset{c}{=} ((passed \rightarrow Drive) \\
&\quad \Box\ (stop \rightarrow MainProc)) \\
GoalSelected &\overset{c}{=} (start \rightarrow Drive) \\
MainProc &\overset{c}{=} (newgoal \rightarrow GoalSelected) \\
\texttt{main} &\overset{c}{=} MainProc
\end{aligned}
$$

$dir : \{-1, 0, 1\}$
$current : \mathbb{Z}$
$goal : \mathbb{Z}$

$true$

Init

$goal = current \land current = Min;\ dir = 0$

enable_*stop*

$goal = current$

effect_*passed*

$\Delta(current)$

$current' = current + dir$

effect_*start*

$\Delta(dir)$

$\neg\ goal > current \lor (dir' = 1);\ \neg\ goal < current \lor (dir' = -1)$

effect_*newgoal*

$\Delta(goal)$

$Min \leq goal';\ goal \leq Max;\ goal' \neq current$

$\neg\Diamond(\updownarrow passed \,;\, (\ell \leq 3) \,;\, \updownarrow passed)$
$\neg\Diamond(\lceil current \neq goal \rceil \,;\, (\lceil current = goal \rceil \land (\ell \geq 2)))$

Abbildung 8: CSP-OZ-DC-Specification „Elevator“

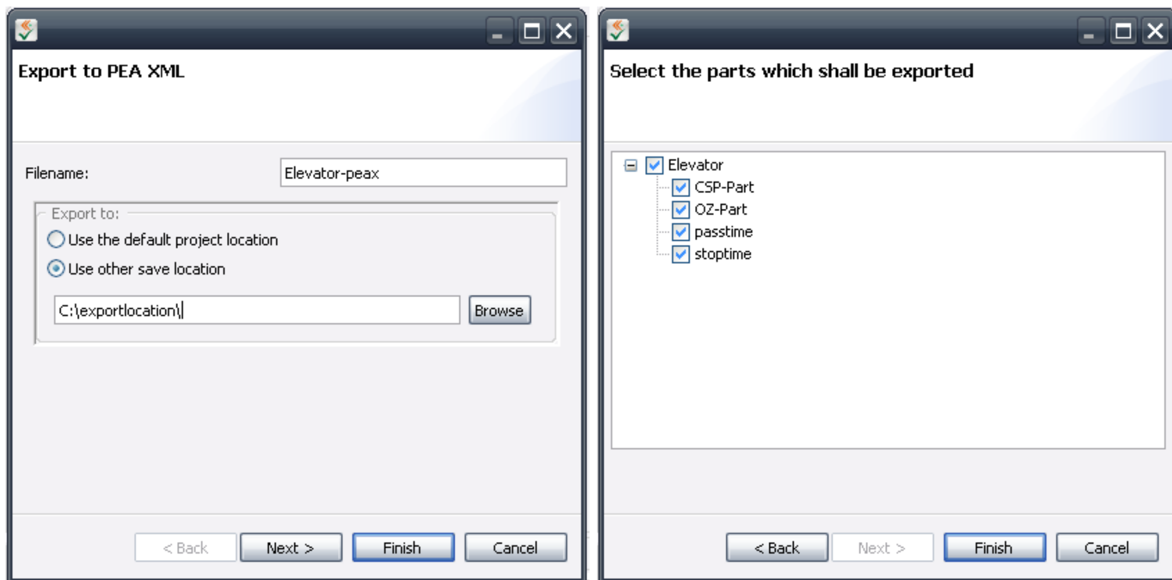

Abbildung 9: PEA-XML-Export

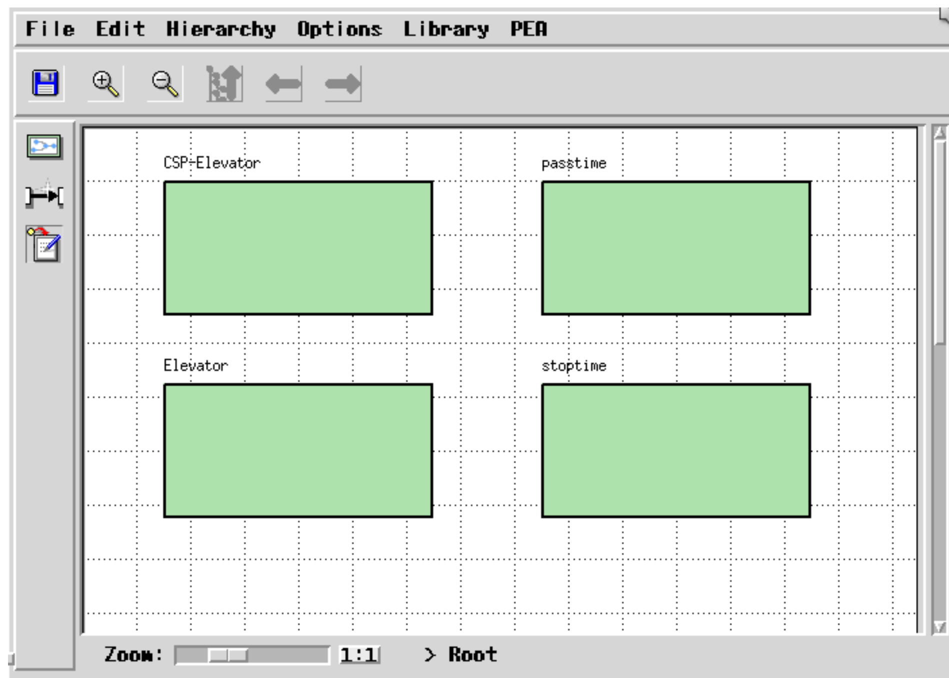

Abbildung 10: Moby/PEA: PEA-System

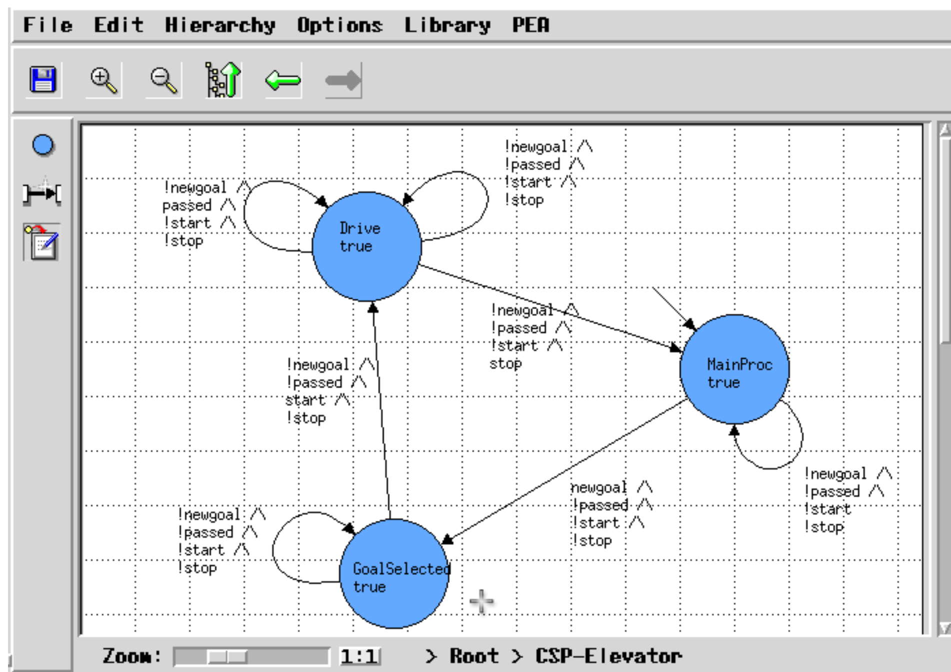

Abbildung 11: Moby/PEA: Verfeinerung CSP-Elevator

4. Entwurf und Implementierung

In diesem Kapitel wird zunächst mal ein Überblick über die Struktur von Syspect gegeben. Dies soll zeigen, wie das neue PEAEditor-Plugin eingefügt werden kann, und welche schon vorhandenen Teile von Syspect wiederverwendet werden können. Daraufhin wird gezeigt, wie das Modell für die Phasen-Event-Automaten in die schon vorhandenen Syspect-Plugins integriert wird. Mit dem eigentlichen Editor und den Funktionalitäten auf dem PEA-Modell beschäftigt sich dann der Rest dieses Kapitels, beginnend mit einer Auflistung aller verwendeten Extension-Points aus Eclipse und Syspect, und mit einer Pakete-Übersicht.

4.1. Verwendete Teile von Syspect

Syspect besteht aus einer Sammlung von Plugins, die in der Regel logisch abgegrenzte Teile des Gesamtsystems bilden. An dieser Stelle wird in kurzer Form auf die Plugins eingegangen, die für den PEA-Editor von Bedeutung sind, sei es durch Erweiterung von Extension-Points, Benutzung einer abstrakten Oberklasse, Erstellung neuer Model-Elemente oder Verwendung diverser Hilfsklassen. Es werden hier noch keine Details für eine Implementierung des PEA-Editors gegeben, sondern es wird erst einmal möglichst allgemein von einem neuen Diagramm-Editor ausgegangen. Zweck dieses Vorgehens ist es, den Entwicklern weiterer Diagramme einen Überblick benötigter Plugins zu geben. Zu einer ausführlichen Beschreibung aller Plugins von Syspect wird auf den Endbericht[6] des Syspect-Projekts verwiesen. In der Produkt-Konfiguration von Syspect werden u.a. das Erscheinungsbild festgelegt und die zu ladenden Plugins ausgewählt. Dementsprechend muss das neue Plugin PEAEditor zu der Liste der Plugins hinzugefügt werden.

SyspectCore Wichtige Klassen für Syspect sind in diesem Plugin enthalten, wie beispielsweise der `UndoManager`, um Benutzer-Aktionen rückgängig zu machen, der `NameAdministrator`, der für eindeutige Namen für Elemente innnerhalb eines Projektes sorgt und das `SyspectUtil`, das einige nützliche Hilfs-Methoden enthält. Um das Prinzip der DeleteActionHooks, welches zum Beispiel dafür sorgt, dass das Löschen von Model-Elementen auch in den jeweiligen Editoren berücksichtigt wird, für alle anderen Syspect-Plugins verfügbar zu machen, bietet `SyspectCore` den Extension Point `syspectdeleteactionhook` an. Plugins, die Diagramm-Editoren enthalten müssen die Extension Points `syspectdiagram` und `syspecteditpartfactory` erweitern. Benutzt ein Plugin die `PreferencePages` von Eclipse, können die Präferenzen beim Start von Syspect auf Default-Werte gesetzt werden, wenn `preferenceinitializer` erweitert wird.

SyspectNavigator Der Navigator stellt der Syspect-Perspektive ein neues View zur Verfügung, in welchem das Modell in einer Baumansicht dargestellt wird. Durch Erweiterung der Erweiterungspunkte `toolbarcontributor` und `actioncontributor` können der Werkzeugleiste des Navigators Aktionen(z.B. „Neues Diagramm erstellen“) hinzugefügt

[6]http://syspect.informatik.uni-oldenburg.de/doc/Endbericht.pdf

werden, bzw. auf „Doppelklicken“ von Elementen im Navigator kann mit dem Öffnen des entsprechenden Diagramm-Editors reagiert werden.

SyspectPersistence Dieses Plugin ist für das Laden und Speichern von Projekten zuständig. Andere Plugins, die neue Syspect-Projekte erstellen, können sich über eine Methode den Default-Speicherort für Projekte geben lassen oder das neue Projekt dem `AutoSaveManager` übergeben.

SyspectUtils Basisfunktionalitäten für ein einhaltliches Layout in allen Syspect-Plugins, sowie allgemeine Widgets und Wizards, werden durch das SyspectUtils-Plugin bereitgestellt. Andere Plugins sollten auf diese zurückgreifen und sie nicht neu implementieren.

Einen wesentlichen Teil der Entwicklungsumgebung Syspect bilden zwei verschiedene Konzepte. Das erste Konzept behandelt alle Elemente der Spezifikation, wie z.B. Klassen, Interfaces, Komponenten, StateMachines usw, und bestimmt die Struktur der Datenhaltung. Dabei werden drei verschiedene Modell-Ebenen unterschieden: *Model*, *Viewable* und *View*. Die *Model*-Ebene beinhaltet die Elemente des UML-Profils, die *Viewable*-Ebene beinhaltet alle Ansichten auf die Elemente und die *View*-Ebene enthält für jedes Ansichts-Element der *Viewable*-Ebene ein rein grafisches Element. Auf der *Viewable*-Ebene wird festgelegt, welche Modell-Elemente (z.B. Attribute, Methoden) auf der *View*-Ebene angezeigt werden sollen. Die View-Ebene bildet die Schnittstelle zum zweiten Konzept, dass die Darstellung und Editierbarkeit der Diagramme mittels des Plugins GEF behandelt. Wie schon im Grundlagen-Kapitel beschrieben, ist setzt das GEF-Plugin das MVC-Pattern um. Die *Model*-Ebene aus Sicht von GEF entspricht der *View*-Ebene aus den drei Syspect-Modell-Ebenen. Die *View*-Ebene von GEF arbeitet mit Figures und anderen Klassen aus dem `Draw2d`-Plugin. Als Verbindungsebene arbeitet die *Controller*-Ebene mithilfe von EditParts und Listener. Die Syspect-Plugins, die diese beiden Konzepte umsetzen, sind die beiden folgenden:

SyspectModel Die Interfaces sind auf verschiedene Pakete aufgeteilt, je nachdem, zu welcher Ebene ihre Elemente gehören und welche Rolle sie in der Struktur spielen (Kernelemente, Zusatzelemente, Verbindungen, etc.). Die Implementierungen dieser Interfaces sind wiederum in entsprechende Pakete unterteilt. Die Erzeugung von Modell-Objekten geschieht nur über die drei Fabriken `ModelFactory`, `ViewPointFactory` und `ViewFactory`, entsprechend dem Factory-Muster [GHJV95]. Neben der Struktur mit den drei Syspect-Modell-Ebenen definieren die Interfaces eine Parent/Child-Hierarchie. Beispielsweise hat der ProjectManager als Kinder verschiedene Projekte, ein Klasse als Kinder Attribute und Methoden. Ein Event-System sorgt dafür, dass die Teile der Struktur über Änderungen informiert werden, die sich dafür interessieren. Plugins, die neue Modell-Elemente zu Syspect hinzufügen, müssen diese, soweit möglich, in die gegebene Struktur eingliedern und das Factory-Muster verwenden.

SyspectDiagramEditorBundle Für alle Diagramm-Editoren stellt das Plugin Basis-Funktionalitäten zur Verfügung. Es bietet Klassenhierachien für `Commands`, `EditParts`, `EditPolicies` und `Figures`. Die speziellen Diagramm-Editoren können auf diese Hierarchien aufbauen und ihren Code hinzufügen. Das `EditorBundle` definiert außerdem die Extension-Points für die `DiagramConsistencyChecker`. Wird ein neuer

Diagramm-Editor implementiert, der auf einen `ConsistencyChecker` zurückgreifen soll, muss im
`SyspectDiagramBundle` ein zusätzlicher Extension-Point definiert werden.

Für beide Konzepte stehen ConsistencyChecker zur Verfügung:

SyspectModelConsistencyChecker Immer wenn Event über die Änderung an einem Projekt aufmerksam macht, beginnt das Plugin, das Modell auf Inkonsistenzen zu untersuchen. Werden welche gefunden, erscheinen diese als Fehler oder Warnungen in der `ProblemView`, einer der Standard-Views von Eclipse. Gegebenenfalls werden die Probleme auch in den jeweiligen Diagrammen visualisiert.

SyspectDiagramConsistencyChecker Für jede Diagramm-Art enthält dieses Plugin einen `ConsistencyChecker`, der Benutzereingaben validiert. Ein Beispiel hierfür ist die Überprüfung, ob eine Verbindung zwischen zwei Knoten hergestellt werden darf. Wird für ein neuen Diagramm-Editor ein `ConsistencyChecker` hinzugefügt, muss dieses Plugin auch den entsprechenden Extension Point aus dem `SyspectDiagramEditorBundle` erweitern.

Die folgenden Plugins werden für die Exporte und Konvertierungen benötigt.

SyspectPEAExport Aus der CSP-OZ-DC-Spezifikation können PEAs als Zwischenprodukt erzeugt werden. Diese können dann weiter ins XML-Format exportiert oder in ein Eingabe-Format für Model-Checker umgewandelt werden.

SyspectCSPOZDCExport Zunächst einmal sorgt dieses Plugin dafür, dass das UML-Modell nach CSP-OZ-DC exportiert wird. Andere Plugins, wie z.B. der SyspectPEAExport können dieses Zwischenprodukt dann weiter exportieren, es kann aber auch direkt als LaTeXoder XML ausgegeben werden. Die Wizards und einige zugehörige Klassen dieses Plugins können von anderen Exportern wiederverwendet werden.

SyspectOZDCUtil Diese Plugin ist ein zentraler Bestandteil von Syspect, da es die Funktionalitäten für Object-Z und Duration Calculus umfasst. Neben den Werkzeugen der *Community Z Tools*, die in dem CZTWrapper gekapselt sind, werden Formel-Editore, Listener und Parser zur Verfügung gestellt.

PEA Tool Das PEA-Toolkit wurde als Plugin in Syspect integriert. Die Funktionalitäten wurden bereits in dem Grundlagen-Kapitel kurz beschrieben. Verschiedene Konvertierungen und Exporte nutzen diese Werkzeuge und verwenden Objekte aus dem Toolkit als Zwischenprodukte. Auch für den PEAEditor bietet sich die Benutzung dieser Funktionen an, um Redundanzen zu vermeiden.

SyspectProperties Die Anzeige der Eigenschaften von Modell-Elementen regelt dieses Plugin mithilfe der *Tabbed Property Views*. Es erweitert drei `Extension-Points`:

- *org.eclipse.ui.views.properties.tabbed.PropertyContributor*

- *org.eclipse.ui.views.properties.tabbed.PropertyTabs*
- *org.eclipse.ui.views.properties.tabbed.property.Sections*

PropertyContributor ist die Schnittstelle zwischen der TabbedPropertyView und allen Workbench-Parts (Navigator, Editor), die auf das Modell zugreifen wollen. Aus diesem Grund müssen neue Diagramm-Editoren das zugehörige Interface `ITabbedProperty-SheetPageContributor` implementieren und außerdem die Methode `getAdaptable(Class adapter)` überschreiben, um Zugriff auf die PropertyView zu erhalten. Die PropertyTabs definieren Reiter für die PropertyView, denen die Sections für die Eigenschaften des selektierten Elements zugeordnet werden können. Hierbei handelt es sich um Widgets, deren Klasse von `AbstractPropertySection` abgeleitet wird. Zur Erzeugung dieser Widgets steht auch die `TabbedPropertySheetWidget-Factory` zur Verfügung, die für ein einheitliches Aussehen sorgt. Um die Anzeige der Eigenschaften aktuell zu halten, wird die Section als Listener beim entsprechenden Modell-Element registriert. Werden Eigenschaften über die PropertyView geändert, wird ein `SimplePropertyChangeCommand` aufgerufen, das mithilfe des Java-Beans-Konzeptes und des Enums `EPropertyConstants` die Attribute des Modells neu setzt. Für neue Modell-Elemente steht eine Vielzahl an Sectionen als Muster zur Verfügung. Die Sectionen können durch die in den Extension-Points definierten ID's an die gewünschte Stelle platziert werden.

SyspectImageExport Für die Diagramme existieren bereits die Export-Formate BMP, EPS, JPEG und PDF. Über das `SyspectPlugin` besorgt sich der Exporter die EditPartFactory für das ausgewählte Diagramm und zeichnet es, statt in den Editorbereich, in eine Datei des gewünschten Formats. Ein neuer Diagramm-Editor auf Basis des `EditorBundle` enthält somit schon automatisch die Funktionalität dieser Exporte.

Syspect enthält bereits drei Diagramm-Arten, die jeweils in einem eigenen Plugin untergebracht sind:

- **SyspectClassDiagramEditor**
- **SyspectComponentDiagramEditor**
- **SyspectStateMachineEditor**

Für die Entwicklung eines neuen Diagramm-Editors bieten diese Plugins einen guten Einblick in die zu verwendenden Konzepte. Die plugin.xml enthält beispielsweise alle benötigten Abhängigkeiten und zeigt die Erweiterung der `Extension Points`, die Verwendung des `SyspectEditorBundle` als Basis für den neuen Editor lässt sich aus den implementierten Klassen nachvollziehen, wie auch allgemein das Konzept von GEF.

4.2. Einbettung von *PEAEditor* in Syspect

Nachdem nun die zahlreichen Plugins aufgezählt wurden, die für die Integration eines neuen Diagramm-Editors, bzw. in einigen Fällen eines speziellen PEA-Editors, benötigt werden, wird nun ein verfeinerter Integrations-Entwurf für den PEA-Editor betrachtet.

4.2.1. Integration des PEA-Modells in das Syspect-Modell

Die Elemente im PEA-Modell, d.h. PEAs, Phasen, Transitionen und Variablen, sollen im Model-Explorer, wie auch im Diagramm-Explorer aufgelistet werden. Dies ist in etwa analog zu dem einer `StateMachine`, die auch in beiden Explorern erscheinen, wobei sie einmal als Diagramm-Kind eines *Projects* und im anderen Fall Kind einer *Capsule* sind. Entsprechend des Syspect-Konzepts mit drei Modell-Ebenen (*Model*, *Viewable*, *View*) muss für ein Element aus dem PEA-Modell, das auch in dem Diagramm erscheinen soll, jeweils ein Element auf jeder der drei Ebenen existieren. Im `SyspectModel` werden diese Elemente jeweils in eigenen Paketen zu den Modell-Interfaces als auch zu den Modell-Implementierungen hinzugefügt. Zusätzlich werden die Fabriken um geeignete create-Methoden ergänzt. Die Modell-Elemente können nicht im `SyspectPEAEditor` abgelegt werden, damit keine Abhängigkeiten von diesem Plugin erzeugt werden (z.B. vom Navigator).

PEA-Klassen PEAs sollen in Containern zusammengefasst werden. Mit der Forderung nach Wiederverwendung bietet sich das Interface `IClass` und seine abstrakte Implementierung `ClassImpl` zur Erweiterung an. Zur Verwaltung der Variablen und Uhrenvariablen kann das `Attribute`-Feld mit den zugehörigen Zugriffsmethoden verwendet werden, wobei Uhrenvariablen den Typen `IClock` bekommen. Die Erweiterung dieser Klassen hat aber vor allem den Vorteil, dass sich damit die `PEAClass` recht einfach in den Navigator einbetten lässt, da Instanzen von IClass dort schon als Kinder der Projekte aufgelistet werden. Es ist lediglich dafür zu sorgen, dass für Instanzen von IPEAClass ein passendes Icon angezeigt wird. Auch für die Anzeige der Eigenschaften in der TabbedPropertiesView verringert sich der Programmieraufwand, da für `IClass`-Instanzen schon *PropertySections* für die `Attribute`-Liste und für die `ZTypeDefinitions` angezeigt werden.

PEA-Projekte Misch-Projekte sind aus im vorherigen Kapitel beschriebenen Gründen nicht erwünscht. Die Klasse `IProject` wird aber aus Gründen der Wiederverwendbarkeit,u.a. bezüglich Persistenz und Navigator, für PEA-Projekte verwendet. IPEAClass als Erweiterung von IClass ist somit ein Kind von IProject.

In einigen Fällen muss zwischen PEA-Projekten und Syspect-Projekten unterschieden werden, um exklusiv „reine“ Projekte zu bewahren, und um auch nur die für den Projekt-Typ erlaubten Aktionen (z.B. Exporte, Verifikation) zuzulassen. Zu diesem Zweck implementieren Instanzen von Syspect-Projekten das leere Interface `ISyspectProject`, das lediglich zur Markierung dieses Projekt-Typs dient.

4.2.2. Verwendete Extensions

Das Plugin-Konzept von Eclipse erlaubt mit Hilfe der Extensions und Extension-Points eine reibungslose Integration von neuen Plugins in eine vorhandene Eclipse-Anwendung. Das PEAEditor-Plugin verwendet folgende von Eclipse zur Verfügung gestellte Extension-Points, die alle das Benutzer-Interface betreffen:

- *org.eclipse.ui.actionSet*

- *org.eclipse.ui.popupMenus*
- *org.eclipse.ui.commands*
- *org.eclipse.ui.bindings*
- *org.eclipse.ui.editors*
- *org.eclipse.ui.preferencePages*
- *org.eclipse.ui.views.properties.tabbed.propertySections*
- *org.eclipse.ui.views.properties.tabbed.propertyTabs*

Die ersten vier Extension-Points behandeln Benutzeraktionen, die über Menüs, Popup-Menüs und Tastaturkürzel gestartet werden können. Durch *editors* wird der neue `PEAEditor` als Editor im Sinne der Eclipse-Workbench kenntlich gemacht und mit den Aktionen verknüpft. Einen eigenen Bereich in den Präferenzseiten der Eclipse-Anwendung erhält das *PEAEditor*-Plugin durch die Erweiterung von *preferencePages*. Die letzten beiden Extension-Points konfigurieren die *Tabbed Property View*. Größtenteils geschieht dies auch schon in dem PlugIn *SyspectProperties*, in Fällen wo ein Objekt durch den Benutzer selektiert wird, das nur im *PEAEditor*-Plugin bekannt ist (z.B. ein Phasen-Knoten im Editor), muss die Konfiguration innerhalb des *PEAEditor*-Plugins erfolgen.

Auch Syspect definiert einige Extension-Points, die schon teilweise im vorigen Abschnitt über die verwendeten Plugins erläutert wurden.

- de.syspect.core.syspectdiagram
- de.syspect.core.syspecteditpartfactory
- de.syspect.core.preferenceinitializer
- de.syspect.navigator.actioncontributor
- de.syspect.navigator.toolbarcontributor

Durch *syspectdiagram* wird der `PEAEditor` als ein Editor für Syspect-Diagramme kenntlich gemacht, *syspecteditpartfactory* stellt die Verbindung zum GEF-Framework her, indem die EditPartFactory festgelegt wird, wie auch das oberste Modell-Element im Diagramm (`IVPhaseEventAutomaton`).
Um die Präferenzseiten von Eclipse mit Standardwerten zu initialisieren, kann *preferenceinitializer* erweitert werden. Mit den letzten beiden Extension-Points wird der *SyspectNavigator* erweitert.

4.3. Das PEAEditor-Plugin

Im Folgenden wird der Aufbau des Plugins gezeigt. Der Inhalt der Konfiguration-Datei *plugin.xml* wurde schon in den vorherigen Abschnitten erläutert, sodass jetzt zunächst die Paket-Struktur betrachtet wird. Da sich das Plugin gut in zwei logische Teile, einem graphischen und einem für Modell-Transformationen, aufteilen lässt, folgt anschließend eine separate Behandlung des Diagramm-Editors und der Konverter.

4.3.1. Paket-Struktur des PEAEditors-Plugins

Die Paket-Struktur wird hier durch Paketdiagramme aus der UML dargestellt und orientiert sich am Aufbau des *SyspectClassdiagramEditors*. Abbildung 12 zeigt die oberste Paket-Ebene. Die Pakete *controller* und *view* haben jeweils noch Unterpakete. Eine komplette Auflistung aller Pakete und ihrer Klassen befindet sich in Anhang A.

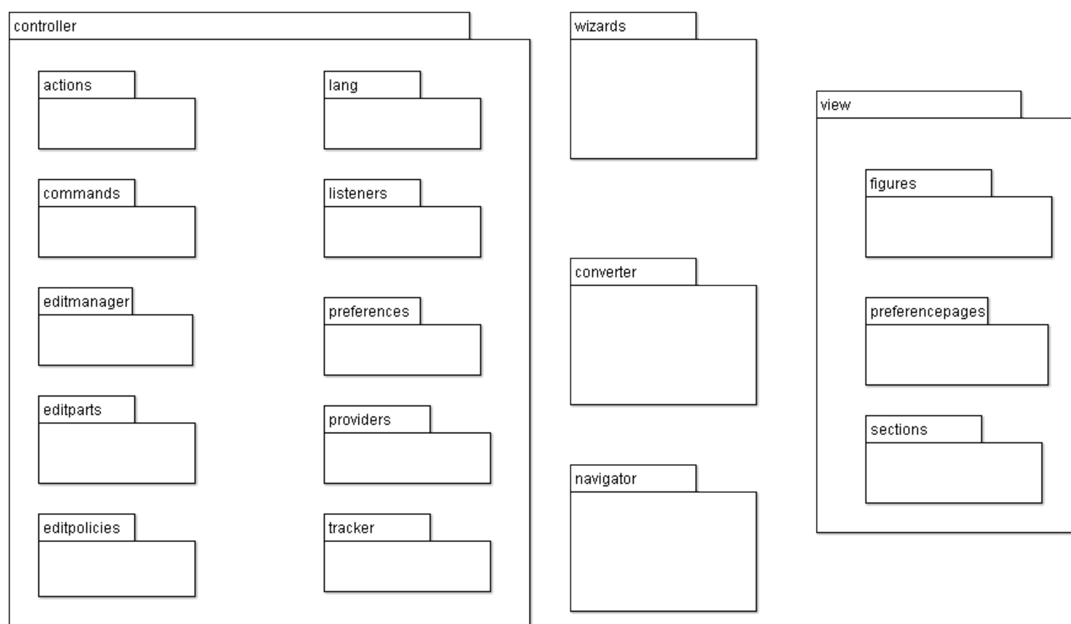

Abbildung 12: Die Paketstruktur

4.3.2. Der Editor für Phasen-Event-Automaten

Der graphische Teil des Plugins spiegelt zum einen die Anforderungen des GEF-Frameworks wider, indem die Klassen erzeugt werden, um das MVC-Pattern umzusetzen, und die dazugehörige Event-Behandlung. Des Weiteren wird das Benutzer-Interface erstellt, hauptsächlich indem die von Syspect vorgegebenen Basisklassen erweitert (z.B. Wizards, Sections) und neue Aktionen hinzugefügt werden.

Da der Aufbau des Editors weitestgehend den Editoren für Klassendiagramme, Statemachines und Komponentediagramme entspricht, und der Entwurf und die Implementierung bereits in dem Endbericht ausführlich beschrieben wurden, werden an dieser Stelle nur besondere, zusätzlich hinzugefügte Merkmale erläutert.

Verschiebbare Phasen-Label Die Beschriftung der einzelnen Phasen kann durch die Invarianten sehr lang werden. GEF sieht die Position der Label nur innerhalb der Knoten-Figure vor, was dazu führt, dass sie entweder abgeschnitten werden, oder die Knoten eine unerwünschte Größe erhalten, um den kompletten Text anzeigen zu können. Verschieden Lösungsansätze, die aus anderen GEF-Anwendungen synthetisiert wurden, erwiesen sich als zu komplex für eine Umsetzung auf dem Syspect-Modell. Beispielsweise könnten die Knoten-Figure und die Text-Figure Teile einer umschließenden Figure sein, deren Ausmaße sich aus den Positionen der einzelnen Figures ergibt. Problematisch ist in diesem Fall die Ermittlung der Endpunkte einer Transition an der Knoten-Figure. Stattdessen wurde die eigene Idee umgesetzt, die Label als eigene Knoten mit in das Diagramm einzubauen. Diese werden anfänglich unter die zugehörige Phase gesetzt, sind unabhängig von der Phase verschiebbar und werden aber beim Verschieben einer Phase synchron mitbewegt.

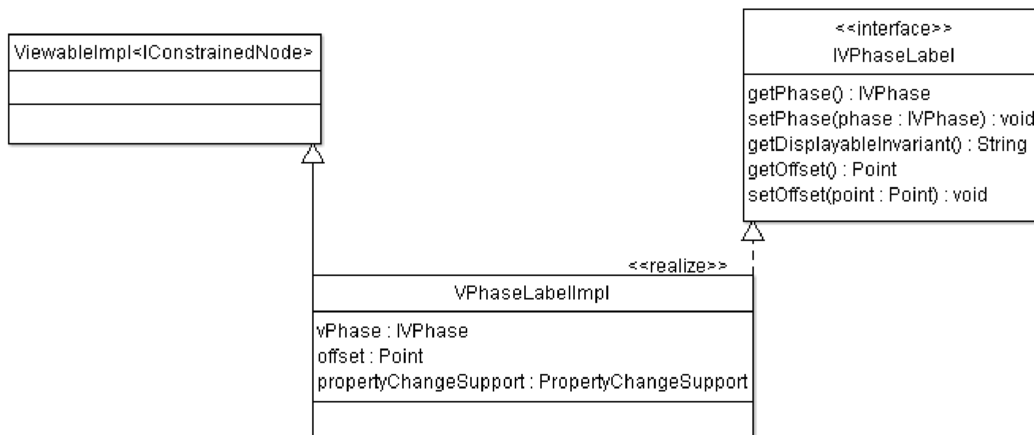

Abbildung 13: VPhaseLabelImpl

Das Viewpoint-Element `VPhaseLabelImpl` (siehe Abbildung 13) erhält hierzu das Feld `offset`, um das Offset zur zugehörigen Phase zu speichern. Der `PhaseLabelEditPart`, der die Rolle des *Controllers* in dem MVC-Pattern erfüllt, wird als Listener beim ViewPoint und beim Modell der Phase eingetragen, um auf Änderungen des Labeltextes und der Position zu reagieren. Als Unterklasse von `GraphicalListenerEditPart` ist er bereits Listener vom Viewpoint des Labels. Die Methode `propertyChange()` unterscheidet eingehende Properties vom Typ `EEventsConstants.CONSTRAINTS_CHANGED` und lässt das Label entweder direkt

mit Hilfe der Methode `setFigureConstraints()` vom `LayoutManager` an die neue Position setzen, wobei `offset` vom Label noch neu gesetzt werden muss (`calcOffset()`) oder sie ruft die Methode `moveLabel()` auf. Hier wird die neue Position der Phase ermittelt und mittels des Offsets die neue Position des Labels ermittelt, welche wiederum an den LayoutManager übergeben wird.

Erstellung einer editierbaren AttributeSection Die `AttributeSection` wird für die Auflistung der Attribute einer IClass-Instanz innerhalb der *Tabbed Property View* bereits verwendet, ist aber mit keiner Editierfunktion ausgestattet. Der PEA-Editor verwaltet die Uhren der PEAs mit Hilfe der `Attribute`-Menge aus der Implementierung der `IPEAClass`. Uhren sollen auch außerhalb der Phasen, in denen sie verwendet werden, hinzugefügt, geändert oder gelöscht werden. Da das Editieren der Attribute auch für die anderen IClass-Erweiterungen sinnvoll ist, wird die alte `AttributeSection` durch eine neue, editierbare ersetzt (Abbildung 14 mit Auswahl an Attributen und Operationen).

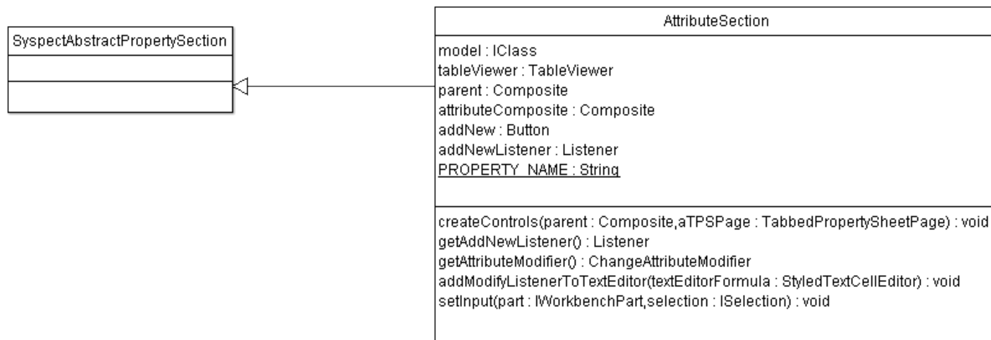

Abbildung 14: Die Klasse AttributeSection

Die Methode createControls() platziert eine Tabelle (`Table`) mit zwei Spalten (`TableColumn` für Name und Z-Typ) und zwei `Buttons` auf einer `Composite`, wobei alle genannten Elemente SWT-Widgets sind. Zu der Tabelle wird ein JFace `TableViewer`-Objekt erzeugt, das mit einem `AttributeContentProvider` und einem `AttributeLabelProvider` verbunden wird. Der Content-Provider muss für den PEA-Editor angepasst werden, da bei diesem die Eigenschaften des Phasen-Event-Automaten, bzw. die der übergeordneten PEA-Klasse, angezeigt werden, wenn der Benutzer auf den Hintergrund (`PEAEditPart`) des PEA-Diagramms klickt. Bei den anderen Diagrammen hat die Selektierung des Hintergrunds keine Auswirkungen.

Um die einzelnen Zellen der Tabelle editierbar zu machen wird jeweils ein `StyledText-CellEditor` für jede Spalte in dem `TableViewer` gesetzt, wobei der für den Z-Typ einen Listener (`StyledTextReplacingListener`) erhält, der Latex-Kürzel in Unicode umwandelt.

Die Methode setInput() ermittelt das zugehörige Modell-Element zur aktuellen Selektion, wobei wieder der Fall des PEA-Diagramm-Hintergrunds berücksichtigt wird. Des Weiteren werden die Buttons mit Listenern versehen (`AddNewAttributeListener`, `DeleteAttributeListener`) und der `TableViewer` erhält als `CellModifier` einen `ChangeAttributeModifier`. Die `Listener` und der `CellModifier` stoßen die Modell-Änderung an, indem sie `ChangeSimple"-PropertyCommands` erzeugen. Der `DeleteAttributeListener` erzeugt genau genommen ein `AttributeDeleteCommand`, das aus mehreren einzelnen `ChangeSimplePropertyCommands` besteht, je nachdem, wie viele Attribute zum Löschen markiert wurden. `ChangeSimplePropertyCommands` haben einen String-Parameter *propertyID*,der die Art des Commands (hinzufügen, löschen) beschreibt, aber auch der neue oder geänderte Wert wird dem String hinzugefügt. Sämtliche Property-IDs sind der Enumeration EPropertyConstants definiert. Das Modell-Element (`ClassImpl`) muss dann in seiner `setPropertyValue()`-Methode die Werte wieder extrahieren. Auch die `getPropertyValue()`-Methode wird für diesen Mechanismus benötigt,denn ein `ChangeSimplePropertyCommand` ermittelt über sie den Modell-Zustand vor der Änderung und speichert diesen für eventuelle Undo-Operationen. Abbildung 15 zeigt ein Beispiel für eine editierbare `AttributeSection`.

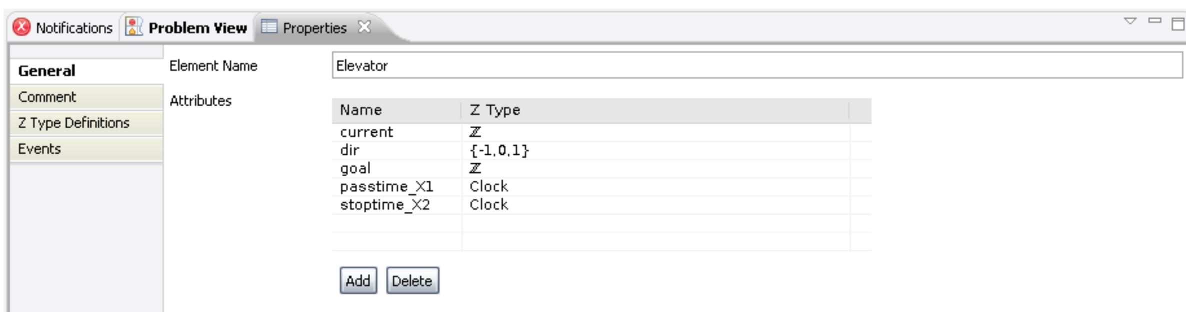

Abbildung 15: Editierbare AttributeSection

Anpassung des automatischen Layouters Das PEAEditor-Plugin ist auf einen automatischen Layouter angewiesen, der die durch Konvertierung erzeugten PEAs in eine benutzerfreundliche und übersichtliche Form bringt. Mit der Klasse `LayoutAction` implementiert Syspect bereits einen Mechanismus für das automatische Layout von Diagrammen, der auf dem `DirectedGraphLayout` von Draw2D basiert. Verwendet wird dieses Layout nur in Klassendiagrammen in vertikaler Ausrichtung, für Automaten ist aber die horizontale Ausrichtung üblich. Auch der kleine Abstand zwischen den Knoten ist für PEAs nicht geeignet, da dort die Transitionen meistens mit langen *Guards* beschriftet sind. Ein weiteres Problem ergibt sich durch die verschiebbaren Phasen-Label, denn auch sie werden von dem vorhandenen Layouter wie normale Knoten behandelt und ohne Bezug zu den zugehörigen Phasen verteilt. Damit dieser Layouter auch für Phasen-Event-Automaten nutzbar ist, müssen einige Änderungen vorgenommen werden. Die Klasse `PEALayoutAction` erweitert `LayoutAction` und überschreibt einige Methoden (Abbildung 16).

Die Richtung des Graphen wird in `setDirection()` auf `PositionConstants.EAST` gesetzt, was horizontaler Ausrichtung entspricht. Für größere Abstände muss von der Methode

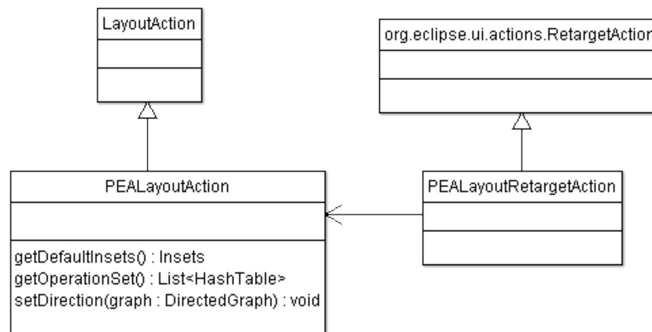

Abbildung 16: PEALayoutAction

`getDefaultInsets()` ein Insets-Object mit höheren Abstandswerten zurückgegeben werden. Um schließlich die Phasen-Label vom automatischen Layout auszugrenzen muss getOperationSet() modifiziert werden, denn diese ermittelt die zusammenhängenden Graphen (und Single-Knoten). Es wird über alle `GraphicalEditParts` iteriert, wobei `PhaseLabelEditParts` dann übersprungen werden. Die `PEALayoutRetargetAction` wird benötigt, um einen Button für die *PEALayoutAction* in die Eclipse-Workbench zu integrieren, der nur dann aktiviert wird, wenn auch das PEA-Editor-Fenster aktiv ist.

4.3.3. Die Konverter

Das *PEA-Toolkit* (PTK), *SyspectPEAExport* und *SyspectARMCExport* stellen bereits sämtliche Funktionalitäten zur Verfügung, um PEAs zu importieren und zu exportieren. Der Export ist möglich in XML und in ein Eingabeformat für ARMC, wobei die Transformation jeweils über PEAs aus dem PEA-Toolkit-Modell geht (PTK-PEAs). Um auf die vorhandenen Funktionalitäten zurückgreifen zu können, und somit den Programmieraufwand zu minimieren, haben wir uns entschieden, die Konvertierung in PEAs aus dem Syspect-PEAEditor (Syspect-PEAs) mit dem Zwischenprodukt der PTK-PEAs durchzuführen, und auch bei dem Export von Syspect-PEAs zu XML oder ARMC zunächst einmal PTK-PEAs zu erzeugen, um anschließend die schon vorhandenen Exportfunktionen aufzurufen. Abbildung 17 stellt dieses Vorgehen grafisch dar.

Daraus ergibt sich, dass zwei Konverter benötigt werden: Ein Konverter von PEA-Toolkit-PEAs zu Syspect-PEAs und einer für die andere Richtung.

PEA-Toolkit-PEA zu Syspect-PEA Die Konvertierung wird durch die `PEAConvertAction` gestartet, welche die vom Benutzer ausgewählten Elemente vom Typ IPEAExportable validiert, die Testformeln aus diesen Elementen ermittelt und einen `PEAConvertWizard` mit diesen Informationen als Parameter erzeugt. Der Wizard besteht aus einer Hauptseite (`PEAConvertWizardPage`), auf der das neue Zielprojekt und die zu übersetzenden Testformeln ausgewählt werden können und aus einer zweiten Seite (`SelectPartsWizardPage` aus dem *SyspectEx-*

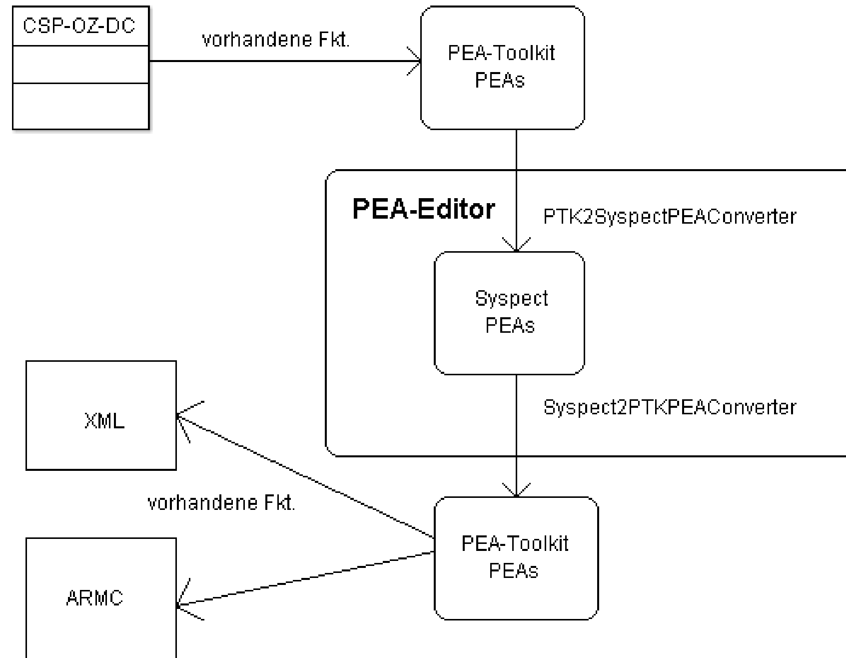

Abbildung 17: Eingliederung der Syspect-PEAs

portBundle), auf der alle CSP-OZ-DC-Teile der selektierten `IPEAExportables` zur Auswahl stehen (Abb. 18). Im nächsten Schritt wird die der PEAConverter erzeugt, der die Transformation der ausgewählten CSP-OZ-DC-Elemente und der Testformeln durchführen soll.

Bei der Analyse der Exporte zu PEA-XML und zu ARMC wurde festgestellt, dass die ersten Schritte bei beiden Exporten mit denen für die Konvertierung benötigten übereinstimmen. Aus diesem Grunde wurde entschieden, die abstrakte Oberklasse `AbstractPEAExporter` zu erzeugen und diese dem Plugin SyspectPEAExport hinzuzufügen. Die wichtigste öffentliche Methode, die `AbstractPEAExporter` implementiert ist `generatePEAs`(), welche PTK-PEAs, PEA-Test-Automaten[7] aus den Testformeln und parallel dazu `PEAInfo` erzeugt.

Die Generierung der `PEAInfo` wurde ursprünglich für die ARMC-Verifikation zum ARMC-Export hinzugefügt [Hob07] und erwies sich auch für die Konvertierung zu Syspect-PEAs als nützlich. Wie bereits erwähnt, gehören die Syspect-PEAs zu einer PEA-Klasse, die als Container dient. Um eine übersichtliche Strukturierung eines PEA-Projektes zu erhalten, und damit man nachvollziehen kann, aus welchen *Capsules* die PEAs generiert wurden, bekommt die PEA-Klasse den Namen der ursprünglichen *Capsule*. Die `PEAInfo` dienen dazu, den Namen bis zur Erzeugung der Syspect-PEA-Objekte und Zuweisung an die PEA-KLassen verfügbar zu machen. Eine als Hilfsklasse zu `AbstractPEAExporter` hinzugefügte `PEAInfo`-Liste sorgt

[7]PTK-PEAs mit Endzuständen

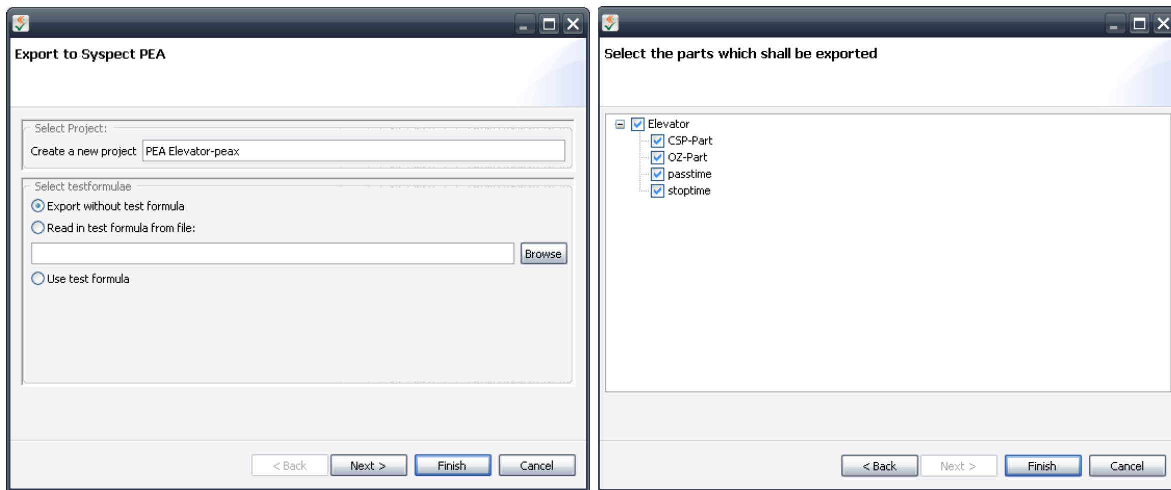

Abbildung 18: Export einer Syspect-Spezifikation (über das Zwischenformat PTK-PEA) in Syspect-PEA

für eine sichere Zuordnung von jeweils einer `PEAInfo` zu einem PEA.

Die Klassen, die `AbstractPEAExporter` erweitern, müssen die Methode `export()` implementieren.

Der `PEAConverter` lässt sich in der *export()*-Methode zunächst die PTK-PEAs mit den zugehörigen PEAInfos übergeben und ruft dann für jeden der Automaten `generatePhaseEventAutomaton()` auf.

Diese Methode bekommt von `getPEAClass()` die zum `PEAInfo` passende PEA-Klasse, die eventuell neu erzeugt wird. Dann werden mit Hilfe der drei Fabriken Objekte von `PhaseEventImpl`, `VPhaseEventImpl` und `DiagramImpl` generiert und die notwendigen Referenzen untereinander gesetzt.

Anschließend werden nacheinander alle Felder des `PhaseEventAutomaton`-Modells generiert, indem jeweils die passende `generate`-Methode aufgerufen wird. `generateDeclaration()` und `generateAttributes()` sind relativ einfache Methoden, generatePhases() ist etwas komplexer. Für alle Phasen im PTK-PEA werden die entsprechenden Syspect-Elemente (*Model*, *View*, *Viewpoint*) durch Fabriken erzeugt. Alle Phasen erhalten zunächst die Position (0,0), da nach Beendigung der Konvertierung ein automatisches Layout erfolgen soll. Der `NameAdministrator` liefert einen eindeutigen Namen. Ist die aktuelle PTK-Phase in der Liste der *Init*-Phasen des PTK-PEAs, wird bei der zugehörigen Syspect-Phase das `Init`-Flag auf „true“ gesetzt. Im Fall, dass der aktuelle PTK-PEA ein `PEATestAutomaton` ist, wird für die Phase geprüft, ob sie in der Liste der `Final`-Phasen enthalten ist und dann das *Final*-Flag der Syspect-Phase entsprechend gesetzt. Für die Generierung der Transitionen werden die PTK- und die Syspect-PEAs (das *Viewpoint*-Element) jeweils zusammen in eine Map geschrieben

Die Transformationen der beiden Invarianten laufen nahezu identisch ab. Die PTK-Phase enthält die Invariante als Constrained Decision Diagram (CDD, [CY05]). Das CDD wird in einen *String* umgewandelt und einem durch die `ModelFactory` erzeugten `ZPredicate` als Prädikat übergeben. In der Syspect-Phase wird die Invariante dann auf das `ZPredicate` gesetzt.

Abschließend wird in für die Phase noch das Label erzeugt und unterhalb der Phase positioniert.

Auch die Methode generateTransitions() ist etwas umfangreicher, wobei auch hier die Syspect-Elemente wieder durch Fabriken erzeugt werden. Die Phasen des PTK-PEAs werden nacheinander betrachtet und die Menge ihrer Transitionen durchlaufen. Im Gegensatz zu Syspect beinhaltet die Menge ein- und ausgehende Transitionen. Damit keine Transition doppelt berücksichtig wird, werden nur ausgehende Transitionen betrachtet. Zu jeder Transition werden die entsprechenden Syspect-Elemente der *Model*-Ebene erzeugt. Das Ziel der Syspect-Transition lässt sich mit der in generatePhases() generierten Phasen-Map ermitteln.

Der *Guard* einer Transition im PEA-Toolkit besteht aus einem CDD für den eigentlichen *Guard*, das mit einem weiteren CDD für die *Events* verknüpft ist. Da in Syspect diese beiden Teile separat behandelt werden, dienen die beiden Methoden `cddToGuard()` und `cddToEvent()` dazu, diese zu trennen. Beide Methoden durchlaufen das CDD, mit dem Unterschied, dass bei `cddToGuard()` alle *Events* ignoriert werden und bei `cddToEvent()` nur die *Events* betrachtet werden. Das Ergebnis der Methoden sind Strings, wobei bei `cddToEvent()` eine Ersetzung aller ZString.SLASH in ZString.NOT stattfindet, um die Negationszeichen in Syspect einheitlich zu halten. Nebenbei werden die nicht-negierten *Events* in der Menge `events` gesammelt.

Die einzelnen Uhren aus der Reset-Menge der PTK-Transition werden als Attribute mit dem ZType „Clock“ in die Attribut-Menge der PEA-KLasse und in die Reset-Menge der Syspect-Transition übernommen.

Nun sind alle Informationen vorhanden, um das Transitions-Label für die Syspect-Transition zu erzeugen. Sind Quelle und Ziel der Transition identisch, wird diese als Schleife mit Hilfe zusätzlicher Bendpoints unterhalb der Phase platziert.

Die gesammelten Events werden durch `generateEvents()` in die PEA-Klasse eingetragen.

Nachdem die Schleife in `export()` fehlerfrei durchlaufen ist, sind für alle PTK-PEAs entsprechende Syspect-PEAs erzeugt worden, die jetzt vom Benutzer weiterbearbeitet werden können. Damit die PEAs optisch etwas ansprechender werden, wird der automatische Layouter verwendet.
Einen durch den Konverter erzeugten PEA, der anschließend automatisch angeordnet wurde, zeigt Abbildung 19.

Syspect-PEA zu PEA-Toolkit-PEA Der Ablauf dieser Konvertierung ähnelt dem aus dem vorigen Abschnitt. Über einen Wizard lassen sich zunächst die Ziel-Datei und die zu konvertierenden Syspect-PEAs auswählen (Abb. 20), dann übergibt der Wizard die PEAs dem `Syspect2PTKConverter`, der die Transformationsschritte durchführt, und abschließend ruft der Wizard die jeweilige, vorhandene Export-Funktion aus Syspect mit den neu erzeugten PTK-PEAs auf.

Die verschiedenen Wizards sind von der Klasse `AbstractPEAExportWizard` abgeleitet (Abb. 21) und implementieren die abstrakte Methode export(), in der die Syspect-Export-Funktionen mit den PTK-PEAs als Parameter aufgerufen werden. Der `AbstractPEAExport-`

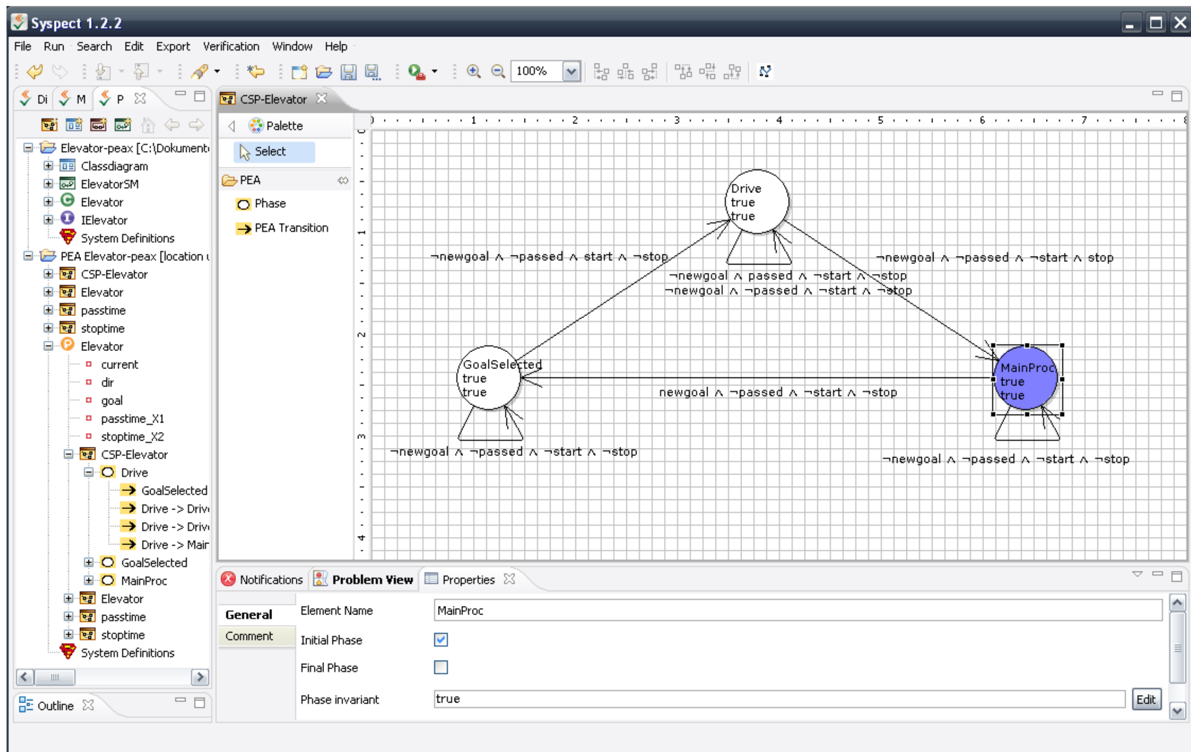

Abbildung 19: Generierter Syspect-PEA

`Wizard` erledigt in der `doFinish`()-Methode bereits durch Aufruf des `Syspect2PTKConverters` die Umwandlung der Syspect-PEAs in PTK-PEAs und generiert die globale *ZTypeDefinition*, die dem Export als weiterer Parameter übergeben wird. Die eigentlichen Konvertierungsschritte im `Syspect2PTKConverter` werden nun genauer betrachtet.

Die Methode convert() des Konstruktors erhält als Eingabeparameter einen Syspect-PEA. Zunächst werden Objekte für die späteren PKT-PEA-Elemente erzeugt. Dies sind u.a. die Menge der Initial-Phasen (`inits`), der Uhren (`clocks`) und der Events (`alphabet`). Die Elemente müssen erzeugt werden, bevor sie später dem Konstruktor für PTK-PEAs übergeben werden, da in der Klasse `PhaseEventAutomata` des *PEA-Toolkits* keine setter-Methoden für eine nachträgliche Änderung existieren. Im nächsten Schritt wird mittels `getChildren()` über die Phasen des Syspect-PEAs iteriert und für jede Phase die Zustands- und die Uhreninvariante erzeugt. Da im PEA-Toolkit die Prädikate als CDD dargestellt werden, müssen die Prädikate der Syspect-PEA in CDDs umgewandelt werden. Für die Zustands- und Uhreninvariante wird dazu die Methode `unicodeToCDD()` verwendet, die Klassen und Elemente des Plugins *SyspectOZDCUtil* benutzt. Anschließend wird eine PTK-Phase, mit dem gleichen Namen wie die Syspect-Phase, und den neu erzeugten Prädikaten generiert. Beide Versionen werden in der *Map* `phaseMap` abgespeichert. Handelt es sich um eine Initial-Phase, wird sie auch in die `init`-Menge aufgenommen.

Nachdem alle Phasen erstellt sind, werden die PTK-Transitionen erzeugt. In allen Syspect-

Phasen werden die ausgehenden Transitionen betrachtet, und für ihre Ziel-Phase die entsprechende PTK-Phase über die `phaseMap` ermittelt. Auch an den Syspect-PEA-Transitionen befinden sich Prädikate, allerdings werden im PEA-Toolkit *Events* und *Guard* zu einem Prädikat *guard* zusammengefasst. Somit wird zunächst ein `CDD` erzeugt für den *Events*-Teil erzeugt und anschließend ein `CDD` für den *Guard*-Teil, der mit and() aus `CDD` mit dem *Event*-CDD verknüpft wird. Die Transformation des *Guard*-Teils wird wieder mittels `unicodeToCDD()` durchgeführt, das *Event*-CDD aber mit der Methode unicodeToEventCDD() generiert, da *Event*-CDDs nur eine Konjunktion von *Events* und keine weitere Prädikatlogik darstellen. Aus der Reset-Menge werden lediglich die Namen der Uhren für die *resets* der PTK-Transitionen benötigt. Die PTK-Transition wird mit `addTransition()` aus `Phase` und mit den zuvor erzeugten Parametern generiert und der Phase hinzugefügt.

Abschließend wird dann ein PTK-PEA mit dem Konstruktor aus `PhaseEventAutomata` erzeugt.

Der Wizard sammelt die einzeln erzeugten PEAs in einer Menge und übergibt diese der `export()`-Funktion zur Weiterverarbeitung.

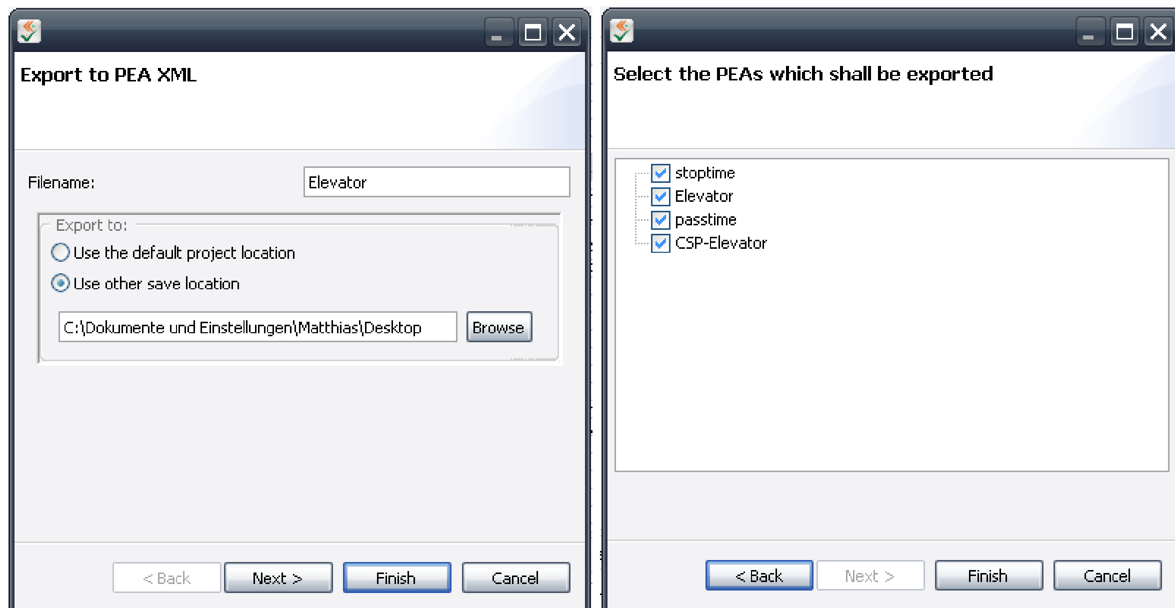

Abbildung 20: Export von Syspect-PEAs nach PEA-XML über das Zwischenformat PTK-PEA

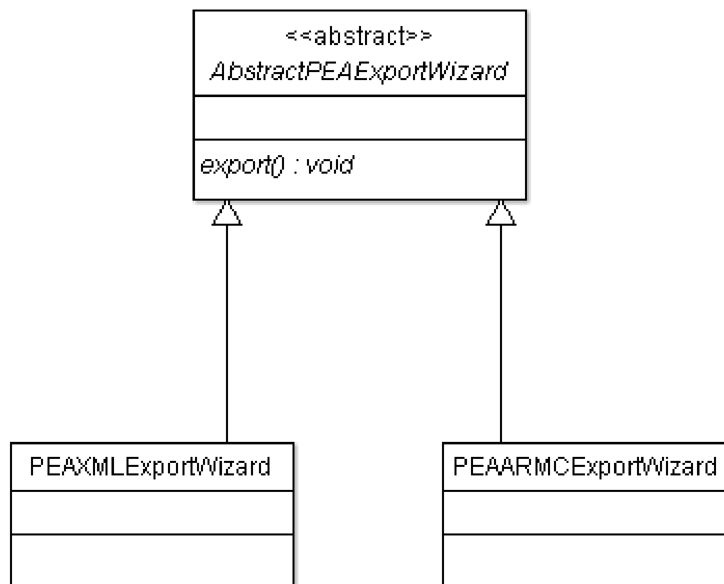

Abbildung 21: Wizards für Exporte von Syspect-PEAs

5. Zusammenfassung und Ausblick

In dieser Arbeit wurde die Integration eines Editor für Phasen-Event-Automaten in Syspect durchgeführt. PEAs lassen sich nun in einfacher Weise erstellen, bearbeiten und in verschiedene Formate konvertieren. Bei der Umsetzung der Programmieraufgabe wurden viele, bereits in Syspect vorhandene, Elemente wiederverwendet, wie Wizards, Basisklassen für Diagramme oder Modell-Elemente und auch Exportfunktionen für Diagramme, u.a. nach JPG, PDF. Einige konnten auch erweitert werden, wie beispielsweise die AttributeSection, ein grafisches Tabellen-Element zur Anzeigen von Klassenattributen, das um die Funktionen zum Hinzufügen, Ändern und Löschen von Attributen ergänzt wurde.

Speziell die Export- und Konvertierungsfunktionen des PEA-Toolkits konnten für die Konverter im neuen PEAEditor-Plugin verwendet werden, was zum einen zusätzliche Redundanz verhinderte, da die gleichen Funktionalitäten nicht noch einmal neu implementiert werden mussten, zum anderen auch für eine höheres Maß an Korrektheit sorgte, da diese Funktionen in der Vergangenheit schon intensiv getestet wurden.

Es ist möglich, von dem PEA-Editor aus Model-Checking zu betreiben, indem die PEAs nach ARMC exportiert werden. Die Benutzung der ARMC-Verifikation in Syspect, bei der gefundene Gegenbeispiele visuell dargestellt werden, ist jedoch noch nicht vorhanden, kann aber mit geringem Aufwand nachträglich implementiert werden.

Der intensive Einsatz des PEA-Editors wird sicherlich für den Wunsch nach einigen Erweiterungen, z.B. zur Erhöhung der Benutzerunterstützung oder zur Verbesserung der Übersichtlichkeit der Diagramme, sorgen. Denkbar sind zusätzliche Wizards, welche Eingaben erleichtern, oder die Möglichkeit, Elemente in den Diagrammen gezielt auszublenden. Aufgrund der in Syspect und diesem Plugin verwendeten Konzepte sind diese Erweiterungen in der Regel leicht implementierbar.

Literatur

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. Cambridge University Press, 1980.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CY05] Kenil C. K. Cheng and Roland H. C. Yap. Constrained decision diagrams. In *AAAI*, pages 366–371, 2005.

[Dau07] Berthold Daum. *Rich-Client-Entwicklung mit Eclipse 3.3. Anwendungen entwickeln mit Eclipse RCP, SWT, Forms, GEF, BIRT, JPA u.a.m.* dpunkt.verlag, 3., überarb., erw. a. edition, 2007.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[FOW01] C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In H. Husmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[HM05] J. Hoenicke and P. Maier. Model-checking of specifications integrating processes, data and time. In J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM 2005*, volume 3582 of *LNCS*, pages 465–480. Springer, 2005.

[HO02a] J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes Data and Time. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer-Verlag, May 2002.

[HO02b] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic Journal of Computing*, 9(4):301–334, 2002. appeared March 2003.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. Communicating sequential processes. Prentice Hall, 1985.

[Hob07] U. Hobelmann. Verifying Properties of Processes, Data, and Time: Linking Counterexamples to High-Level Specifications. Master's thesis, University of Oldenburg, 2007.

[Hoe06] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.

[JHF+94] He Jifeng, C. A. R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. Provably correct systems. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 288–335, London, UK, 1994. Springer-Verlag.

[Lam83] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[LR09] Bernhard Lahres and Gregor Rayman. *Objektorientierte Programmierung - Das umfassende Handbuch*. Galileo Computing, 2009.

[Mey05] R. Meyer. Model-Checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten. Master's thesis, University of Oldenburg, 2005.

[MORW04] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 267–286. Springer-Verlag, March 2004.

[MORW08] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a formal method into a software engineering process with uml and java. *Formal Asp. Comput.*, 20(2):161–204, 2008.

[OMG03a] Object Management Group OMG. Unified modeling language 1.5 specification. omg document formal/2003-03-01, March 2003.

[OMG03b] Object Management Group OMG. Unified modeling language infrastructure, version 2.0. adopted specification, omg document ad/2003-03-01, July 2003.

[Ree79a] T. Reenskaug. Models-views-controllers. *Technical note, Xerox PARC, December*, 1979.

[Ree79b] T. Reenskaug. Thing-model-view-editor, 1979.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[RW03] H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams: Classes and State Machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 229–243. Springer, 2003.

[Smi00] G. Smith. The object-z specification language. Kluwer Academic Publisher, 2000.

[Spi92] J. Michael Spivey. The z notation: a reference manual (2nd edition ed.), 1992.

[SR98] Bran Selic and Jim Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime Limited, 1998.

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb, and Matthias Lübke. *Die OSGi Service Plattform. Eine Einführung mit Eclipse Equinox*. dpunkt.verlag, 2008.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *IPL*, 40(5):269–276, 1991.

A. Pakete und Klassen

A.1. Plugin SyspectPEAEditor

de.syspect.peaeditor

- `Activator`

de.syspect.peaeditor.controller

- `PEAContextMenuProvider`
- `PEAEditor`
- `PEAEditorActionBarContributor`
- `PEAEditorInput`
- `PEAEditorPalette`
- `PEAEditPartFactory`
- `PEAIconStorage`
- `PhaseLabelConstraint`

de.syspect.peaeditor.controller.actions

- `PEAConvertAction`
- `PEACreateAction`
- `PEACreateUndoableAction`
- `PEALayoutAction`
- `PEALayoutRetargetAction`
- `PEATransitionCreateAction`
- `PEAXMLExportAction`
- `PhaseCreateAction`

de.syspect.peaeditor.controller.commands

- `ConnectionBendpointCreateCommand`
- `ConnectionBendpointMoveCommand`
- `PEAEventDeleteCommand`

- `PEATransitionCreateCommand`
- `PEATransitionDeleteCommand`
- `PEATransitionReconnectCommand`
- `PEATransitionWithWizardCreateCommand`
- `PhaseCreateCommand`

de.syspect.peaeditor.controller.editmanager

- `PEATransitionLabelEditManager`

de.syspect.peaeditor.controller.editparts

- `PEAEditPart`
- `PEATransitionEditPart`
- `PEATransitionLabelEditPart`
- `PhaseEditPart`
- `PhaseLabelEditPart`

de.syspect.peaeditor.controller.editpolicies

- `ConnectionBendPointEditPolicy`
- `PEALabelMovePolicy`
- `PEATransitionConnectionEditPolicy`
- `PEATransitionDirectEditPolicy`
- `PEAXYLayoutEditPolicy`
- `PhaseGraphicalNodeEditPolicy`

de.syspect.peaeditor.controller.lang

- `Messages.java`

de.syspect.peaeditor.controller.listeners

- `ChangeEventModifier`
- `EventCreateListener`
- `EventDeleteListener`

de.syspect.peaeditor.controller.preferences

- PreferenceConstants
- PreferenceInitializer

de.syspect.peaeditor.controller.providers

- EventsContentProvider
- EventsLabelProvider
- SelectPEAsContentProvider
- SelectPEAsLabelProvider

de.syspect.peaeditor.controller.tracker

- PhaseLabelTracker

de.syspect.peaeditor.converter

- IPEAConverter
- PTK2SyspectPEAConverter
- PEATermToEventCDDVisitor
- Syspect2PTKPEAConverter

de.syspect.peaeditor.ep.navigator

- DoubleClickContributor

de.syspect.peaeditor.view.figures

- PEAFigure
- PEATransitionFigure
- PEATransitionLabelFigure
- PhaseFigure

de.syspect.peaeditor.view.preferencepages

- PEAPreferencePage
- PEAWizardPreferencePage

de.syspect.peaeditor.view.sections

- EventsSection

de.syspect.peaeditor.wizards

- `AbstractPEAExportWizard`
- `GetSelectedPEAs`
- `PEAConvertWizard`
- `PEAConvertWizardPage`
- `PEAXMLExportWizard`
- `SelectPEAsWizardPage`

A.2. sonstige Syspect-Plugins

In diesem Abschnitt werden Klassen aufgelistet, die in den anderen Syspect-Plugins erstellt oder modifiziert wurden.

de.syspect.controller.eclipse *SyspectCore*

- `IconStorage`

de.syspect.view.eclipse *SyspectCore*

- `SyspectActionBarAdvisor`

de.syspect.consistency.diagram *SyspectDiagramConsistenceChecker*

- `PhaseEventAutomatonConsistencyChecker`

de.syspect.editorbundle *SyspectDiagramEditorBundle*

- `Activator`
- `ExtensionManager`

de.syspect.editorbundle.controller.actions *SyspectDiagramEditorBundle*

- `LayoutAction`

de.syspect.editorbundle.controller.consistency *SyspectDiagramEditorBundle*

- `IPhaseEventAutomatonConsistencyChecker`

de.syspect.editorbundle.controller.lang *SyspectDiagramEditorBundle*

- `Messages`

de.syspect.exportbundle.wizards *SyspectExportBundle*

- `ExportablePartContentProvider`

de.syspect.model.accessory *SyspectModel*

- `AttributeImpl`

de.syspect.model.connections *SyspectModel*

- `ConnectionLabelImpl`

de.syspect.model.core *SyspectModel*

- `ClassImpl`

de.syspect.model.factory *SyspectModel*

- `ModelFactoryImpl`
- `ViewFactoryImpl`
- `ViewPointFactoryImpl`

de.syspect.model.lang *SyspectModel*

- `Messages`

de.syspect.model.pea *SyspectModel*

- `PEAClassImpl`
- `PEAEventImpl`
- `PEATransitionImpl`
- `PEATriggerImpl`
- `PhaseEventAutomatonImpl`
- `Phase`

de.syspect.model.viewpoint *SyspectModel*

- `VPEATransitionImpl`
- `VPhaseEventAutomatonImpl`
- `VPhaseImpl`
- `VPhaseLabelImpl`

de.syspect.modelinterface *SyspectModel*

- `EEventsConstants`
- `EPropertyConstants`

- `ModelUtil`

de.syspect.modelinterface.factory *SyspectModel*

- `IModelFactory`
- `IViewFactory`
- `IViewPointFactory`

de.syspect.modelinterface.pea *SyspectModel*

- `IPEAClassImpl`
- `IPEAEventImpl`
- `IPEATransitionImpl`
- `IPEATriggerImpl`
- `IPhaseEventAutomatonImpl`
- `IPhase`
- `ISyspectPEAExportable`

de.syspect.modelinterface.viewpoint *SyspectModel*

- `IVPEATransition`
- `IVPhaseEventAutomaton`
- `IVPhase`
- `IVPhaseLabel`

de.syspect.navigator *SyspectNavigator*

- `ContentProvider`
- `IconStorage`
- `LabelProvider`
- `View`

de.syspect.export.pea.controller.actions *SyspectPEAExport*

- `AbstractSelectionPEAExportableExportAction`

de.syspect.export.pea.controller.lang *SyspectPEAExport*

- `Messages`

de.syspect.export.pea.controller.wizards *SyspectPEAExport*

- `TestFormulaeGroup`

de.syspect.export.pea.converter *SyspectPEAExport*

- `CSP2PEAConverter`
- `DC2PEAConverter`

de.syspect.export.pea.info *SyspectPEAExport*

- `DCPEAInfo`
- `ObjectZPEAInfo`
- `PEAInfo`
- `PEAPEAInfo`
- `StateMachinePEAInfo`

de.syspect.syspectproperties.controller.commands *SyspectProperties*

- `AttributeDelete`

de.syspect.syspectproperties.controller.listeners *SyspectProperties*

- `AddNewAttrributeListener`
- `ChangeAttributeModifier`
- `DeleteAttributeListener`
- `OZPropertiesChangeKeyListener`
- `PhaseTypeButtonCheckedListener`
- `PropertiesChangeKeyListener`

de.syspect.syspectproperties.controller.providers *SyspectProperties*

- `AttributeContentProvider`

de.syspect.syspectproperties.lang *SyspectProperties*

- `Messages`

de.syspect.syspectproperties.view.sections *SyspectProperties*

- `AttributeSection`
- `ClockInvariantSection`

- `FinalPhaseSection`
- `InitPhaseSection`
- `PEATransitionEventSection`
- `PEATransitionGuardSection`
- `PEATransitionOZSection`
- `PEATransitionResetsSection`
- `PhaseInvariantSection`
- `PhaseOZSectionSection`

de.syspect.syspectproperties.view.wizards *SyspectProperties*

- `SelectResetsSection`
- `SelectResetsWizardPage`