Evaluation of Model Execution Tools

Bachelorarbeit

Yusuf Can Kesen 0627752

Inhalt

[1 Introduction 3](#_Toc346719694)

[2 Problem description 3](#_Toc346719695)

[3 Implementation 4](#_Toc346719696)

[Tabelle 1-Bewertungsschema 9](#_Toc346719697)

[Tabelle 2-Ausführbare Aktionen 10](#_Toc346719698)

[3.1 Cameo Simulation Toolkit 11](#_Toc346719699)

[3.1.1 Ausführung 11](#_Toc346719700)

[3.1.2 Debugging 13](#_Toc346719701)

[3.1.3 State Machine 14](#_Toc346719702)

[3.1.4 Aktivitätsdiagramm 15](#_Toc346719703)

[3.1.4.1 Kommunikation zwischen Engines 15](#_Toc346719704)

[3.1.5 Validierung 16](#_Toc346719705)

[3.1.6 Mock-Up 17](#_Toc346719706)

[3.1.7 Parametrics Simulation 17](#_Toc346719707)

[3.1.8 Mathematisches Engine 17](#_Toc346719708)

[3.1.9 Vorteile & Nachteile 18](#_Toc346719709)

[3.2 IBM Rational Rhapsody Developer 18](#_Toc346719710)

[3.2.1 Ausführung 18](#_Toc346719711)

[3.2.2 Statecharts 20](#_Toc346719712)

[3.2.3 Aktivitaetsdiagramme 22](#_Toc346719713)

[3.2.4 Validierung & Debugging 23](#_Toc346719714)

[3.2.5 Mock-Up 24](#_Toc346719715)

[3.2.5 Unterschied zur Andere Versionen 24](#_Toc346719716)

[3.2.7 Vorteile & Nachteile 25](#_Toc346719717)

1 Introduction

2 Problem description

3 Implementation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ausführbare Funktionalitäten** | **Cameo**  **Simulation Toolkit** | **Ibm**  **Rational**  **Rhapsody** | **Ibm**  **Rational**  **Software Architect** | **Enterprise**  **Architect**  **10.0.1006** |
| **Ausführbare Diagramme** |
| Activitydiagramm | Yes | Yes | Yes | Yes |
| Zustandsdiagramm | Yes | Yes | Yes | Yes |
| Sequenzdiagramm | Aut. generated | Aut. generated | Yes | Yes |
| Andere.. | Parametric Simulation | SysMl | BPMN | BPMN |
| **Zustandsdiagramm** |
| Execution | W3C SCXML Engine | Code Generation | Code Generation | Engine |
| Events on transitions |
| Call Event | No | Triggered Operation | Just internal | No |
| Signal Event | Yes | Yes | Yes | Yes |
| Time Events | Yes | Yes | No | No |
| Change Events | No | Yes | No | No |
| Signals have attributes? | Yes | Yes | Yes | Yes |
| Outgoing transitions from parent element can be triggered | Yes | Yes | Yes | No |
| States |
| State | Yes | Yes | Yes | Yes |
| Orthagonaler State | Yes | Yes | Yes | Yes |
| UntermaschinenState | Yes | Yes | Yes | Yes |
| Composite states | Yes | Yes | Yes | Yes |
| Pseudo States |
| Verzweidung | Nein | Yes | Yes | Yes |
| Verbindung | Nein | Yes | Yes | Yes |
| Initial | Yes | Yes | Yes | Yes |
| Final | Yes | Yes | Yes | Yes |
| Terminate | Nein | Ja |  | Yes |
| Entry/Exit point | Nein | Ja |  | Yes |
| Flacher History | Ja | Nein |  | No |
| Tiefer History | Ja | Ja |  | No |
| Junction | Nein | Ja |  | Yes |
| Guards on Transitions | Yes | Yes | Yes | Yes |
| Connection Point Reference | Nein | Ja? |  | No |
| Manuelles Auflösen der Signale | Yes | Yes | Yes | Yes |
| Can communicate with |  |  |  |  |
| Activity | Yes | Yes | Yes | Yes |
| Statemachine | No | Yes | Yes | Yes |
| Interaction | No | No | Yes | No |
| Actions |  |  |  |  |
| On Transition | Yes | Yes | Yes | Yes |
| Entry | Yes | Yes | Yes | Yes |
| Do | Yes only 1 | Yes | Yes | No |
| Exit | Yes | Yes | Yes | Yes |
| Executable Behaviours from StateMachine Actions |  |  |  |  |
| State Machine | Ja | Indirect | Indirect | Yes |
| Activity | Ja | Indirect | Indirect | Yes |
| Interaction | Nein | No | No | Yes |
| Opaque Behaviour | Ja | Yes | Yes | Yes |
| Must have a classifier? | No | Yes | Yes | No |
| Can Define Behaviour of |  |  |  |  |
| Klasse | Yes | Yes | Yes | Yes |
| Operation | Yes | No | No | Yes |
| **Activity Diagramm** |
| Execution | fUML Engine | Code generierung | Code Generation | Engine |
| Actions |
| Object Node | Yes | Yes | Yes | Yes |
| Datastore/Centralbuffer | Yes | Yes | Yes | Yes |
| Ordering of Object Nodes | Fifo  Lifo  Ordered  Unordered | Nein |  |  |
| Input/Output expansion Node (arrays) | Nein? | Nein |  |  |
| Activity Parameter Node | Ja | Nein |  |  |
| Events on ControlFlow? |
| Time event | Yes | No | No |  |
| Signal Event | Yes | No | No |  |
| Change Event | No | No | No |  |
| Call Event | No | No | No |  |
| Other.. | Guards | Guards | Guards |  |
| Pins |
| Input/output pins | Yes | Only in C++ | No |  |
| Value pin | Yes | No | No |  |
| Activity pin | Yes | Only in C++ | No |  |
| Other elements |
| Initial Node | Yes | Yes | Yes | Yes |
| Activity End Node | Yes | Yes | Yes | Yes |
| Flow Final Node | Nein | Nein |  |  |
| Join/Fork Nodes | Yes | Yes | Yes | Yes |
| Decision/Merge Nodes | Yes | Yes | Yes | Yes |
| Objektfluss | Yes | Yes | Yes | Yes |
| Kontrollfluss | Yes | Yes | Yes | Yes |
| Exception Handler | Nein?? | Nein?? |  |  |
| Unterbrechungsbereich | Nein?? | Nur C++ |  |  |
| Gruppierung von Parametern | Nein | Nein | No | No |
| Partititons/swimlines | Yes | Yes | Yes | Yes |
| Schachtelung von Aktivitaten | Yes | Yes | Yes | Yes |
| Executable Behaviours from Activity Diagramm Actions |
| Activity | Yes | Indirect | Indirect | Yes |
| Opaque Behaviour | Yes | Yes | Yes | Yes |
| State Machine | Yes | Indirect | Indirect | Yes |
| Can Define Behaviour of |  |  |  |  |
| Klasse | Yes | Yes | Yes | Yes |
| Operation | Ja | Nur C++ | No | Yes |
| **Execution Animation** | Yes | Yes | Yes | Yes |
| Level | 3 | 2 | ? | 1 |
| Geschwindigkeit einstellbar | Yes | No | No | Yes |
| **Mock-Up Gui** | Yes | Yes | No | Yes |
| **Debugging** |
| Breakpoints | Ja | Ja | Yes | Yes |
| Debugging Levels | Trace Debug  Info  Warn  Err  Fatal  (Log4j) | Info  Warn  Err |  |  |
| Laufzeit werte der Variablen ansehen | Yes | Yes | Yes | Yes |
| Laufzeit werte der Variablen ändern | Primitive datatypes | Nein | Primitive datatypes | Primitive datatypes |
| Runtime values in math Engine exportpieren & importieren | Ja | No | No | No |
| Variablen in Plot  darstellen | Ja | No | No | No |
| Runtime Objekt exportieren | Ja | No | No | No |
| Simulation Log erstellen | Ja | Ja |  |  |
| Console Log  Erstellen | Ja | Ja |  |  |
| Schrittweise ausführen | Ja | Ja | Yes | Yes |
| Pausieren | Ja | Ja | Yes | Yes |
| Block ausführen | Ja | Ja | Yes | Yes |
| **Ausführung&Simulation** |
| Mehrere Simulation Sessions gleichzeitig | Ja | No | Yes | No |
| Execution Configuration | Yes | Yes | Yes | Yes |
| Manuelles Auflösen  der Signalen | Yes | Yes | Yes | Yes |
| Generierung SequenzDiagramm vor/während der Ausführung | Ja | Ja | No | No |
| Änderung der Diagramm waehrend der Ausführung | No | No | No | No |
| **Validierung** |
| Aut. Validierung vor der Ausführung | Ja | Ja | Yes |  |
| Simulation Validierung | Ja | Ja | Yes |  |
| Fehler angezeigt?Trotzdem ausführbar | Ja/Ja | Ja/Nein | Yes/Yes |  |
| **Opaque Expressions** |
| Expressions | Guards  Constraints  Values  Behaviours  Decisions | Guards  Constraints  Values  Behaviours  Decisions |  |  |
| Sprachen | JavaScript  Beanshell  Groovy  Jruby  Jython  Ocl  Java Binary | Java  C  C++  Ada | UAL  (Alf 1. Level confirmation) | JavaScript |
| Sprach Extension | JSR233 kompatibel | No | No | No |

# Tabelle 1-Bewertungsschema

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No** | **Vordefinierte Executable Actions** | **Cameo**  **Simulation Toolkit** | **Ibm**  **Rational**  **Rhapsody** | **Ibm**  **Rational**  **Software Architect** | **Enterprise**  **Architect** |
| **1** | AcceptCall | Ja | Nein |  |  |
| **2** | AcceptEvent | Ja | Ja |  |  |
| **3** | AddStructuralFeat.Value | Ja | Nein |  |  |
| **4** | AddVariableValue | Nein | Nein |  |  |
| **5** | BroadcastSignal | Nein | Nein |  |  |
| **6** | CallBehavior | Ja | Nein |  |  |
| **7** | CallOperation | Ja | Nein |  |  |
| **8** | ClearStructuralFeature | Ja | Nein |  |  |
| **9** | ClearAssociation | Ja | Nein |  |  |
| **10** | ClearVariable | Nein | Nein |  |  |
| **11** | ConditionalNode | Ja | Nein |  |  |
| **12** | CreateLink | Ja | Nein |  |  |
| **13** | CreateLinkObject | Ja | Nein |  |  |
| **14** | CreateObject | Ja | Nein |  |  |
| **15** | DestroyLink | Ja | Nein |  |  |
| **16** | DestroyObject | Ja | Nein |  |  |
| **17** | ExpansionRegion  Not supported but works! | Ja/Expansion Node not supported | Nein |  |  |
| **18** | LoopNode | Ja | Nein |  |  |
| **19** | OpaqueAction | Ja | Nein |  |  |
| **20** | RaiseException | Nein | Nein |  |  |
| **21** | ReadExtent | Ja | Nein |  |  |
| **22** | ReadİsClassifiedObject | Ja | Nein |  |  |
| **23** | ReadLink | Ja | Nein |  |  |
| **24** | ReadLinkObjectEnd | Nein | Nein |  |  |
| **25** | ReadLinkObjectEndQual. | Nein | Nein |  |  |
| **26** | ReadSelf | Ja | Nein |  |  |
| **27** | readStructuralFeature | Ja | Nein |  |  |
| **28** | readVariable | Nein | Nein |  |  |
| **29** | ReclassifyObject | Ja | Nein |  |  |
| **30** | ReduceAction | Ja | Nein |  |  |
| **31** | RemoveStructuralFeat.Value | Ja | Nein |  |  |
| **32** | RemoveVariableValue | Nein | Nein |  |  |
| **33** | ReplyAction | Nein | Nein |  |  |
| **34** | SendObject | Ja | Nein |  |  |
| **35** | SendSignal | Ja | Ja |  |  |
| **36** | StartClassifierBehaviour | Ja | Durch Rhapsody API |  |  |
| **37** | StartObjectBehaviour | Ja | Nein |  |  |
| **38** | SequenceNode | Nein | Nein |  |  |
| **39** | StructuredActivityNode | Ja | Nein |  |  |
| **40** | TestIdentityAction | Ja | Nein |  |  |
| **41** | UnmarshallAction | Nein | Nein |  |  |
| **42** | ValueSpecification | Ja | Nein |  |  |
| **43** |  |  |  |  |  |

# Tabelle 2-Ausführbare Aktionen

# 3.1 Cameo Simulation Toolkit

Cameo Simulation Toolkit wurde von der Firma No Magic als ein Plug-In für die MagicDraw Modelling Tool, entwickelt. Die Simulationsframework basiert auf Subset for Executable UML Models (fUML) und State Chart XML (SCXML) Standards. Cameo Simulation Toolkit beinhaltet das Ausführen von Modellen und das Validieren eines Systemverhaltens durch das Ausführen, Debuggen und Animieren von State Machines und Activity Diagrams in UML 2.0 Standarten. Das Tool ermöglicht auch erstellen und durchführen eine realistische GUI für modellierte Systeme. Sequenzdiagrammen können zwar generiert werden aber ausführen von Sequenzdiagrammen ist nicht implementiert.Es besteht die möglichkeit eine Interaktionsengine zu API hinzufügen.Ausserdem wird Alf(UML Action Language) wird nicht unterstütz.

# 3.1.1 Ausführung

Um ein Element ausführen zu können, muss dieses Element von einer der existierenden Engines im Cameo Simulation Toolkit unterstützt werden.Diese Elemente sind:

* Aktivitäten
* Aktivitätsdiagramme
* State Machine
* State Machine Diagramm

Eine Klasse oder eine Instanz Spezifikation von dieser, deren Klassifikatorverhalten als eine Aktivität oder ein State Machine definiert ist, kann ausgeführt werden. Die Simulationen werden ohne Code-Generieung, mit Engines ausgeführt. Cameo Simulation Toolkit verfügt verschiedene Engines zum Ausführen von verschiedenen Elementen, sowie Aktivitäten und State Machines. Das ausgewählte Behaviour wird im dazugehörigen Diagramm ausgeführt. Wenn eine Klasse oder eine Instanz ausgeführt wird, wird ein “runtime value” durch dieses Element erstellt, um die simulierten Werte dieser Klasse speichern zu können.

Als Alternative bietet das Cameo Simulation Toolkit ein Stereotyp (ExecutionConfig) für die Erstellung einer Ausführungskonfiguration. Durch diese Konfiguration können mehrere Eigenschaften der Ausführung eingestellt werden. Für eine Instanz oder Klasse können mehrere Konfigurationen erstellt und direkt ausgeführt werden. Wichtige Eigenschaften dieser Konfiguration sind:

* Einordnung der Benutzeroberfläche für die Ausführung
* Auswählen von auszuführenden Elementen
* Einstellen von Animationen
* Erstellung eines Logs für die Ausführung
* Einstellen eines Ausführens von Automatisch/Direkt von Elementen
* Einordnen von executionListeners ,die die Events einer Ausführung empfangen; z.B.: automatische Erstellung eines Sequenzdiagramms
* Einstellen des Verhältnisses zwischen echter Zeit und Simulationszeit

Für die weitere Analyse der Ausführung Cameo Simulation Toolkit bietet sich die Möglichkeit eine Execution Log durch Stereotype “ExecutionLog” zu generieren, um alle in der Laufzeit geschehenen Ereignisse zu speichern. In dieser Log können mehrere Simulation Sessions gespeichert werden. Die Laufzeit Ereignisse, die in Log je nach Wahl gespeichert werden können, sind:

* Ausgewählte Werte einer Klasse (Laufzeit-Werte)
* Signale
* Behaviour oder Operation Aufrufe
* Sequenz der Aktivierung und Deaktivierung
* Constraint Failures

Die Stereotype ausgewählte “Execution Log” muss in der Stereotype “ExecutionConfiguration” unter dem Punkt “log” ausgewählt werden, um die ExecutionLog zu erstellen.

Die Modelle und Diagramme können vor der Ausführung automatisch validiert werden. Die exisitierenden Fehler werden mit Erklärungen angezeigt.

Die Simulationen eines Elements in eine State Machine, in ein Aktivitätsdiagramm und die Ausführung kann anhand von verschiedenen Farben nach Auswahl animiert werden. Die Animationsgeschwindigkeit ist einstellbar. Es gibt drei Zustände, in dem sich ein Element befinden kann: “im aktiven Element”, “zuletzt besuchtes Element” und “besuchtes Element”.

# 3.1.2 Debugging

Cameo Simulaton Toolkit ermöglicht das Debugging während der Ausführung von Elementen. Bei der Ausführung eines Elements wird eine neue Simulations Session generiert. Die generierte Simulations Session beinhaltet einer Instanz des ausgeführten Elements. Die Laufzeit Werte werden dann in dieser Instanz geschrieben. Für jeden Aufruf eines anderes Verhaltens (State Machine, Aktivität) wird eine neue Session erstellt.

Für alle Elemente eines ausführbares Diagrammes können Breakpoints hinzugefügt werden. Die Ausführung wird dann pausiert, wenn ein aktives Element einen aktiven Breakpoint hat. Die Framework ermöglicht den Debugging Funktionen beim Ausführung sowie Pausieren und schrittweisen Ausführen.

Die Informationen werden während der Ausführung in einer Simulation Konsole angezeigt. Es gibt sechs verschiedene Informationsstufen (Log4j) :

* Trace
* Debug
* Info
* Warn
* Error
* Fatal

Die anzuzeigenden Informationsstufen sind einstellbar. Für die spätere Informations Analyse der Simulation, werden diese logs automatisch gespeichert.

Das Laufzeit-Objekt, sowie die Variablen von dem Objekt werden im Variablen-Fenster angezeigt. Es ist möglich die Variablen mit den Typen Integer, String, Boolean und Real während der Ausführung zu ändern. Die Änderungen der numerischen Variablen können auch in einem Plot (Wert/Ausführungszeit) dargestellt werden.

Das am Simulationsanfang generierte Laufzeit Objekt aus der Kontextklasse kann als eine neue Instanz-Spezifikation exportiert oder auf eine bestehende Instanz-Spezifikation überschrieben werden.

# 3.1.3 State Machine

Cameo Simulation Toolkit benutzt die ein Open-Source Apache Implementation W3C SCXML (State Charts XML) Engine, um die State Machines zu simulieren. Dabei wird kein Code für die Ausführung generiert. Die W3C SCXML Standart verfügt über eine State Machine basierte Ausführungsumgebung basierend auf Harel Statecharts.

Die State Machine Ausführungsfähigkeit des Tools umfasst die meisten Elemente der StateMachine Diagramm Spezifikation von UML 2.0. Eine Liste der unterstützen Elemente sind in Tabelle 1 unter dem Punkt StateMachine zu sehen.

Um die Transitionen aufzulösen, werden Trigger benutzt. Die Trigger können entweder Signal, Time Events oder Guards sein. Wenn keine Trigger gegeben sind, wird eine Transition nach der Ausführung von alle Aktivitaten in dem Source Zustand automatisch durchgefürt. Die selbstausgehenden Transitionen müssen ein zugewiesenes Signal oder Time Event besitzen. Zusätzlich können Signal Events Attribute beinhalten.

Die Signale können manuell in Simulationsbar ausgelöst werden, durch Empfangen von Signalen, durch eine Aktivität oder mittels erstelltem GUI durch das Zuweisen eines Signals zu einem Gui Element. Alle Signale, die empfangen werden können, können durch das Auswählen des Zielzustandes im Variablen-Fenster aufgelistet werden. Es ist auch möglich automatische Signale von einer Aktivität mit einer “Send Signal Action”, zu einer State Machine zu senden.

Die Zustände müssen einen Namen haben und können je eine Entry, Exit und Do Aktivität haben, die ausgeführt werden können. Do Aktivitäten werden nur einmal ausgeführt. Aus Entry/Exit/ Do Aktivitäten und Transitionen kann als Verhalten ein StateMachine, eine Aktivität oder ein opakes Verhalten aufgerufen werden. Ein Verhalten weist genau einen Besitzer auf. Wenn ein Verhalten durch eine andere do/enter/exit Aktivität oder von einer Transition aufgerufen wird (und nicht durch seinen Besitzer),wird automatisch ein neues Aktvititätsdiagramm generiert. Diese generierte Aktivität enthält dann eine CallBehaviourAction, die wieder dieses Verhalten als sein Verhalten beinhaltet.

Für das Ausführen eines opaken Verhaltens wird die ScriptEngine benutzt. Die Liste der Sprachen, die unterstützt werden, sind in Tabelle 1 unter dem Punkt “Opaque Expressions” zu finden.

Vor oder während der Ausführung ermöglicht das Tool eine automatische oder manuelle Erstellung eines Sequenzdiagrammes. Nachdem das Sequenzdiagramm generiert ist, werden alle aufgelösten Signale automatisch in diesem Sequenzdiagramm hinzugefügt.

Es können verschaltete States (composite State), Orthogonale States und (Submachinen) Untermaschinen States simuliert werden. Jedoch werden Connection Point Reference, Verzweidung, Verbindung, Terminate and Entry/Exit Knoten nicht in der Simulation unterstützt.

Falls sich der aktive State in einem Substate befindet und es eine ausgehende Transition, mit dem gleichen Signal, ausgehend von Substate gibt, wird zuerst die Transition, die von der active State in Substate ausgeht, aufgelöst. In orthogonalen States gilt auch die gleiche Semantik. Falls in einem orthogonalen State die Signale von beiden aktiven Zuständen ausgehenden Transitionen gleich sind, wird eine der aktiven Zustände zufällig ausgewählt und einen Schritt weiter ausgeführt, und danach folgt der Andere. Die flachen und tiefen History States werden unterstützt.

# 3.1.4 Aktivitätsdiagramm

Cameo Simulation Toolkit simuliert Aktivitätsdiagramme und Aktivitäten mit OMG fUML Standart Engine und für die Ausführung wird kein Code generiert. Die Engine ist ein ausführbares Subset der Standart Uml Spezifikation und führt die Simulation durch Interpretation von auszuführende Modell duch. Die Engine verfügt auch durch eine fUml 1.0 Specification Unterstüztung.

Die Aktivitätsdiagramm-Ausführung unterstützt zahlreiche Elemente der Uml 2.0 Spezifikation. Eine vollständige Liste der unterstützten Actions für die Ausführung ist in der Tabelle 2 und die anderen Elemente in Tabelle 1 zu finden.

Opakes Action sowie Entscheidungsknoten, Guards und Constraints können in die aufgelisteten Sprachen impletiert werden, die in Tabelle 1 unter dem Punkt Sprachen, ersichtlich sind.

# 3.1.4.1 Kommunikation zwischen Engines

Die Kommunikation zwischen Aktivitätsdiagramm und State Machines wird durch SendSignalAction erstellt. Aktivitaetsdiagramm können zu State Machines signale Senden und diese Signale können dann in State Machines Transitionen auslösen.

Aktivitatesdiagramme können ausserdem mit Send und Receive Signal Actions miteinander kommunizieren.

Mit CallBehaviourAction kann ein Aktivitats Diagramm ein State Machine oder ein andere Aktivitaetsdiagramm aufrufen. Mit CallOperationAction ist es möglich eine State Machine oder ein Aktivitaetsdiagramm aufzurufen wenn sie verhalten einer Operation definieren.

StateMachines können auch andere StateMachines oder Aktivitaetsdiagramme durch ihre entry/do/exit afrufen.

# 3.1.5 Validierung

Cameo Simulation Toolkit bietet eine automatische Simulation-Validation von Modellen vor der Ausführung. Dabei werden die fehlerhaften Verbindungen ,Assoziationen, Konfigurationen, nicht unterstütze Elemente (z.B.:Unmarshall Action, AddVariableValueAction... ) im Simulation-Validierung Fenster und im Kontainer Baum gekennzeichnet. Die Validierungsergebnisse sind als Informationsstufen in 5 Kategorien eingeteilt :

* Info
* Debug
* Warning
* Err
* Fatal

Ausserdem wird der Kontext des Fehlers und die Message des Fehlers ausgegeben. Falls es in dem Projekt Fehler gibt, die höher als “warning” eingestuft sind, wird das Modell nicht ausgeführt, da diese Fehler den Ablauf der Simulation stark beeinflussen oder gar nicht ausführbar sind. Die Fehler, die in weniger als “warning ” eingestuft sind (z.B.: Behaviour of a CallBehaviourAction is not specified) müssen nicht korregiert werden, da die Ausführung trotzdem erfolgen kann.

# 3.1.6 Mock-Up

Das Tool bietet ein schnelles und einfaches Erstellen eines GUIs mittels User Interface Modelling Diagramm. Wenn diese erstellte GUI dann optional in ein “Execution Configuration” als GUI für “Execution” selektiert wird, wird die Ausführung des Verhaltens durch GUI erfolgen.

Unterstütze Elemente für das Erstellen eines simulierbaren GUIs sind:

* Frames (stellt einen Klassifikator dar)
* Panels (kann andere GUI Elemente oder wieder ein Panel behalten)
* Gruppe(um andere Elemente zu gruppieren)
* Labels, check boxes, text fields, sliders

(können die Laufzeit-Werte eines primitiven Types darstellen)

* Buttons (um Signale zu senden oder um eine/ein Operation/Verhalten aufzurufen)
* combo boxes, spinners , radio buttons (können Enumarationen darstellen)

# 3.1.7 Parametrics Simulation

Cameo Simulation Toolkit hat die Parameter Engine Plug-In um ein mathematisches Modell eines komplexen Systems kalkulieren zu können. Diese Systeme, in dem die Parameter Simulation durchgeführt werden kann, werden in SysML, eine auf UML basierende Sprache für das Modellieren von Komplexen Systemen, modelliert.

Die Parametrics Engine benutzt die Mathematische Engine um die Berechnungen durchzuführen. Dafür kann entweder Built-in Math Engine oder Matlab Engine herangezogen werden.

# 3.1.8 Mathematisches Engine

Cameo Simulation Toolkit hat einen Built-in Math engine, die bereits für Parametrische Simulation benutzt wird. Die Engine kann auch seperat für komplexe mathematische und logische Berechnungen benutzt werden.

Die Laufzeit-Werte einer Instanz können auch für weitere Evaluationen direkt nach Math Eingine exportiert werden oder ein definierter Wert in Math Engine kann von einer Property einer Instanz importiert werden.

# 3.1.9 Vorteile & Nachteile

Wird bearbeitet...

+Nutzen des Diagramme auch in Funktion/Operation ebene

+Modelle können auch ohne Kontext simuliert werden

+Umfangreiche Aktivitaetsdiagramm implementierung

-Kein methodenaufruf auf Transitionen bei StateMachines

# 3.2 IBM Rational Rhapsody Developer

Rational Rhapsody Developer ist eine Produkt der Rational Rhapsody Family der Firma IBM. RRD ermöglich den Entwicklern die Entwicklung von Embedded oder Echtzeit Systeme auf der Basis der UML modellierung. RRD generiert ANSI conform code in sprachen Java, C, C++ und ADA von der aufgebauten Diagramme. Durch die generierte code können die Statecharts und die Aktivitaetsdiagramme simuliert werden. Ausserdem durch die Sequenzdiagramme können die nachtrichten austausch in der simulierten System visualisiert werden.

# 3.2.1 Ausführung

In Rhapsody können Statecharts und Aktivitaetsdiagramme, Verhalten einer klasse definieren. Zusatlich in C++ version die Aktivitaetsdiagramme können auch funktionsverhalten definieren. Um eine Diagramm zu Simulieren ,muss dieser Diagramm verhalten einer Klasse definieren, da die Diagramme nur in diese kontext simuliert werden kann.

Einer der key feautures der Rhapsody ist Multi-threading. Es gibt 2 Arten von Klassen, aktiv und sequentiell. Aktive Klassen haben ihre eigene Thread.Rhapsody generiert eine neue Thread für aktive Klassen sobald sie initalisiert werden. Sequentielle Klassen dagegen laufen in RJIMainThread(Main Thread der Ausführung).

Unter Ausführung Konfigurationen es kann eingstellt werden ,Wleche Klassen in Main thread initalisiert wird, welche compiler für die compilation benutzt wird und welche Art von Ausführung(Trace siehe Validierung & Debugging und Animation) durchgeführt wird.Danach wird zuerst durch Rhapsody code erzeugt und nachher wird die code von jeweilige compiler compiliert. Rapsody hat keine integrierte Compiler, die pfad der jeweilge Compiler muss angegeben werden. Nachdem das Projekt “gebuilt” ist die Ausführung kann mit “Run Executable” kommando gestartet werden. Falls der Art der ausführung “Animation” ist ,werden nur dann die Ausführbare diagramme animiert.Bei Trace version Rhapsody stellt eine Kommandozeile für die kommandoeingabe für die Ausführung.

Um die nachtrichten wahreand der Ausführung zu speichern wird ein Sequenzdiagramm animiert.Daher muss der Benutzer zuerts selber eine Sequenzdiagramm mit gewahlte Klassen als Instancelines aufbauen.Diese Diagramm wird dann automatisch basis für die animierte Sequenzdiagramm von Rhapsody übernommen.

Danach wenn das “Go” Button gedrückt wird, werden die gewahlte Klassen initalisiert und ihre Klassenverhalten werden animiert. Die nachtrichtenaustausch zwischen Klassen und der Benutzer kann in Animierte Sequenzdiagramm beobachtet werden.

Rhapsodu verfügt Object Model Diagramme um die statistische Struktur des simulierten Systems aufzubauen.Object Model Diagramme sind gleichzeitig Klassen und Objekt Diagramme und damit kann die assoziationen zwischen den Elemente aufgebaut werden. Durch diese aufgabaute Struktur können die Klassen andere Klassen oder Objekten erkennen und zu denen Namen geben. Durch diese namen können die Objekte einander erkennen und kommunizieren.

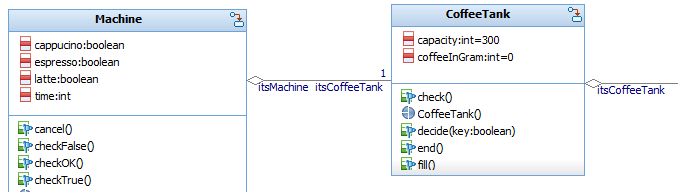


Figure 1. Object Model Diagramm

zB.: Klasse Machine kann in Klasse CoffeTank mit ihr Name und Rhapsody API definierte gen() methode ein Signal Auslösen:

itsCoffeTank.gen(new fill()); (siehe 3.2.2 Statecharts).

# 3.2.2 Statecharts

Für die Animation und ausführung der Statecharts wird code von der aufgebauten Statechart generiert. Statecharts können in RRD die laufzeit verhalten einer Klasse definieren. Für die Statecharts die, die verhalten von primitive Operationen definieren wird kein code generiert.

Statecharts können auf die Transitionen als Trigger Signal Events, Change Events, Time Events und durch Rhapsody definierte “Triggered Operations” auch Call Events haben.Guards auf Transitionen können mit Call und Signal Events kombiniert werden.

Rhapsody Statecharts benutzt Events für asynchrone communication und triggered Operations für synchrone communication. Gesendete Events werden in der Zielobjekt in eine Eventqueue gespeichert und werden ausgeführt wenn sie dran sind.Triggered operations dagegen blockieren ihre absender bis die empfanger Statechart auf die triggered operation antwortet.Ausserdem kann null Transitionen auch benutzt werden.

Zustande haben Entry Action, Exit Action und Iinterne Transitionen(Do). Es kann beliebeig viel interne Transitionen hinzugefügt werden, die wieder mit en Event ausgelöst werden können. Verscahltete und Orthogonale States werden unterstütz. Rhapsody Statecharts unterstützen ausserdem Send und Accept Event Action und Accept Time Event.

Mit Send Action können Statecharts zu andere Statecharts oder Aktivitaets diagramme Events senden und mit Accept Action auch diese empfangen(nur in Java,C,C++).Eine andere möglichkeit mit andere Statecharts zu interargieren ist durch die Rhapsody API zu verfügung gestellte gen() methode.Mit gen() methode können Events und triggered Operations in das selbe oder in eine andere Statechart ausgelöst werden.Mit dieser Methode kann auch ein Statechart Transition aus eine Aktivitatestdiagramm ausgelöst werden oder ein Statechart kann zu einer Aktivitaetsdiagramm ein Event senden.

zB.:Eine Server Klasse,die mit ein Statechart oder Aktivitaetsdiagramm definiert ist, kann in einer Client Klasse, die mit eine Statechart oder Aktivitaetsdiagramm definiert ist mit der aufruf

” itsClient.gen(new eventX()); “

ein Event (eventX) auslösen.Wobei “itsClient” die definierte name der Client Klasse für die Serverklasse ist.

In Rhapsody Statecharts die Zustands oder Transaktions Aktionen können mit Java, C, C++ oder ADA code definiert werden. Von Aktionen kann ein Statechart oder Aktivitaetsdiagramm nicht direkt aufgerufen werden, da in Rhapsody Statecharts und Aktivitaetsdiagramme nur als Verhalten von einer Klasse definierbar sind(in C++ version auch als funktionsverhalten für Aktivitaetsdiagramme siehe 3.2.4 Aktivitatsdiagramme). Indirekt ist es möglich von eine Aktion eine Statechart oder Aktivitaetsdiagramm aufzurufen. Mit der code “itsX.startBehavior();” Verhalten einer Klasse kann aufgerufen werden wobei “itsX” die definierte name für die Klassifikator der aufgerufene Verhalten ist.

Obwohl primitive Operationen nicht als trigger benutzt werden, primitive Operationen einer Klasse können von andere Klasse aufgerufen werden solange diese Klassen zueinander in Beziehung stehen.

Wenn in eine verschaltete State zwei Transitionen in konflikt stehen oder wenn ein Event zwei Transitionen auslösen kann, Rhapsody gibt die vorrang zu in der niedrigere schicht stehende Transition.Bei eine Orthogonale State in so einem fall ist es keine vorrang regel definiert.

# 3.2.3 Aktivitaetsdiagramme

Rhapsody benutzt auch code generation für Ausführung der Aktivitaetsdiagramme. Aktivitaetsdiagramme können verhalten einer klasse und nur in Rhapsody Developer for C++ version auch funktions verhalten definieren.

Wenn ein Aktivitaetsdiagramm als funktionverhalten definiert ist, Rhapsody generiert die code für diese Diagramm in eine neue Functor Klasse, die diese diagramm als Klassenverhalten implementiert. Die Aufgabe diese funktion zu ausführen(eine funktion die mit eine Aktivitatsdiagramm assoziert ist) ist dann zu eine neue Functor klasse abgeordnet. Wenn diese funktion aufgerufen wird, wird dann die Klassenverhalten der Functor aufgerufen.

Somit wird die code generation eigenschaft für Klassen verhalten benutzt um code für eine Funktionsverhalten zu generieren.

Klassen können mehr als einen Aktivitaetsdiagramm haben aber nur eine davon darf als Klassifikator verhalten zugeordnet sein.

In vergleich zu Statecharts, Aktivitatsdiagramme sind nicht sehr detaliert implementiert in Rhapsody. Die ausführbare vordefinierte Aktionen sind Send, Accept Event und Accept Time Event Actions.Call Behaviour und Call Operation Action sind zwar vorgegeben aber dafür wird kein code generiert. Die verhalten von Call Operation oder Call Behaviour action kann aber mit Rhapsody API (“startBehavior” wie in Statecharts siehe Statecharts 3.2.3) definiert werden. Ausserdem es st möglich primitive Operationen anderer Klassen aufzurufen wenn diese zu Aufrufer Klasse in Beziehung stehen.

Mit Send Action oder durch Rhapsody API definierte “gen()” methode(ident wie Statecharts) können Aktivitaetsdiagramme zu andere Aktivitaetsdiagramme oder Statecharts Events und triggered operations senden oder aufrufen.Mit Accept Event Action können diese Events von eine andere Aktivitatesdiagramm oder Statechart empfangt werden.

Verschaltete Actions oder Subactivities werden unterstütz. Actions verhaten können durch Java,C,C++ oder Ada code definiert werden.Object Knoten können variablen oder Instanzen von Klassen definieren und wird in der code generation untersütz. Input und Outputs pins ,Activity Parameter und Interruptable Region wird nur in C++ version unterstützt.

Transitionen dürfen keine Triggers besitzen aber zuweisung eines Guards zu transitionen ist möglich. Zahlreiche Elemente das Aktivitatesdiagramm wird untersütz bei code generation (siehe Tabelle 1).

# 3.2.4 Validierung & Debugging

Rhapsody verfügt eine interne Checkliste(Components>Configurations>Host)um die korrektheit des ausführbare diagramme zu überprüfen. Durch Rhapsody gefundene Fehler in Diagramme werden in “Check Model” fenster angezeigt.Rhapsody bietet auch erstellen von benutzerdefinierte externe Cheks mit Rhapsody API. Ein modell kann immer mit “Check Model” kommando auf interne oder externe kriterien kontrolliert werden.Um die Code Generation erfolgreich durchführen zu können müssen alle interne Check kriterien erfüllt werden. Fehler, die durch Cheks gefunden werden sind, werden in 3 kategorien eingestuft:Fehler, Warnung und Informaiton.

Für kontrollieren der code ist der jeweilige compiler der Sprache des codes zustandig. Die von Compiler gefundene fehler werden in “Build” Fenster angezeigt.

Nach dem “Generate” kommand wird zuerst das Modell kontrolliert und falls es Fehler vorliegt werden diese angezeigt. Danach mit “Make” kommando wird das code compiliert und falls vorhanden die fehler aus der Compiler angezeigt. Falls dasProgramm nicht kompilierbar ist wird keine .exe datei erzeugt.

Nachdem laufzeit Instanzen initalisiert sind können breakpoints für debugging definiert werden.Dabei kann der benutzer die Ausführung schrittweise ausführen oder pausieren. Dabei stellt Rhapsody für verfügung verschiedene Listeners für Debugging zb.:State entered, State exited, Action ready, Action Done, Instance created etc..

Ausserdem Instanzen jeder Klasse können in Browser Fenster unter Ordner “Instances” gesehen werden.Dabei stehen folgende informationen zur verfügung:

* Wieviele Instanzen existieren von einer Objekt.
* Beziehungen einer Instanz mit andere Instanzen
* Laufzeit werte der Atrribute von Instanzen

Die werte derAttribute können nicht geandert werden.

Für weitere analyse kann die ausführung als Trace gestartet werden.Dafür muss die punkt Instrumentation, unter Konfiguraiotenen für ausführung, muss statt Animation auf Trace gesetzt werden. Rhapsody stellt eine Tracing API zu verfügung mit zahlreiche kommandos. Mit Hilfe dieser kommandos kann der Ausführung sehr ohne user-interface ,durch konsole kontrolliert werden.Obwohl die Ausführungsfaehigkeiten bei GUI und Kommando gleich sind , bei Trace modus können detalierter information über dem status der ausführung empfangt werden.

Eine andere möglichkeit für weitere analyse ist die animated Sequenzdiagramm in dem alle mögliche information sowie alle Events und Zustandsaenderungen, über der ausführung gespeichert wird.

# 3.2.5 Mock-Up

Rhapsody bietet Panel Diagramme um eine GUI für beobachtung und kontroll einer Applikation. Panel Diagramm eignet sich eher für Kontroll von Maschinen basierte Applikationen(embedded Systems) mit imitieren Hardware Elemente für Mschinen so wie Drehknöpfe , Led and Matrix Displays , on/off Switch Button sowie Anzeigeninstrumente. Ausserdeö wird Slider,Text Boxen und Buttons wird unterstütz.Mit dieser Elemente können Signale gesendet/ausgelöst oder werte angezeigt und geandert werden. Ausführen eines GUI wird nur in der versionen für C und C++ unterstützt.

# 3.2.5 Unterschied zur Andere Versionen

Wird bearbeitet...

Rational Rhapsody Designer for Systems Engineers version ist eine model-based systems engineering (MBSE) umgebung, das auch ermöglicht modellieren und simulieren von Sys-ML Modellen.

# 3.2.7 Vorteile & Nachteile

Wird bearbeitet...

+Multithreading

+Embedded/Echtzeit systeme modellieren

+Erzeugt einsetbare programme, nutzbare code

+Synchron/Asynchron Kommunikation

+Kommunikation zwischen Modelle durch Rhapsody API

+Umfangreiche Statechart Implementation

-Schwache Aktivitaetsdiagramm implementation(wenig vordefinierte Aktionen)

-Statecharts können nur Klassenverhalten definieren und keine Operations

Altivitaets diagramme nur bei C++ version.

-no operation behaviour

+ sequence diagram

# 3.3 IBM Rational Software Architect

Ibm Rational Software Architect is a product of Ibm Rational Family and offers an extension “Simulation Toolkit ” to simulate Uml Models. RSA is build up on Eclipse IDE Framework.

RSA is capable of simulating Activities defined by Activity and Interaction Diagrams, State Machines defined by Statechart Diagrams and Interactions defined by Sequence Diagrams. All expressions and Action Behaviours in Diagrams can be defined using UML Action Language(UAL). UAL is an implementation of Alf minimum conformance level in RSA . Additionaly other languages such as Java and C++ can be used as action language. RSA simulates the models by interpreting the generated source code.

3.3.1 Execution & Debugging

There are 2 kinds of simulation types in RSA. Analysis simulation is used in the early stages of the development. Behaviours can be simulated without having classifiers. The disadvantage of this kind of simulation is that the behaviours can not communicate with each other, since the communication is established over objects which are instances of UML classifiers. In this case, all signals must be send manually.

In UAL simulation, behaviours such as Activities and State machines must have UML classifiers such as an UML component or an UML Class in order to communicate with each other. The communication can be established direct through the object references or indirect using ports and and connectors, which can be defined in a composite structure diagram. More about the communication will be discussed later.

The only behaviour that can be defined by a statemachine, activity or a sequence diagram is the classifier behaviour. Operation behaviours can not be defined by such behaviours but with UAL code. This causes some restrictions since in complex models activities defines mostly operation behaviours. The problem can be solved by initializing an object and using its classifier behaviour for every single call for an operation, which is not that optimal .

Rsa provides a rich execution and debugging environment and can run multiple sessions simultaneously. Behaviours can be simulated with “Execute as - Model” command in the context menu of the diagram or the classifier. If more than one behaviour is defined for a classifier, the behaviour that should be executed must be selected. Classifier which have references of other classifiers as constructor parameter, can not be simulated directly. This kind of behaviour must be initialized with the UAL code first.

After initalizing the classifier, the UAL methods “**init()”** and “**start()”** should be called to initialize and start the classifier behaviour.

A simulation can be launched as a “normal” or “random ” execution. In random mode execution, the decisions such as decision points with guards, more outgoing transitions from one state and the order of sent signals will be made random.

After executing the “Execute As” command RSA changes the perspective to “Model Execution View”(an extended debug perspective of Eclipse IDE ) which contains Debug, Variables, Breakpoints, Events and Console views.

Debug views contains commands such as resume, suspend, step into or terminate and lists the active simulation sessions including all run-time instances(of UML classifiers), received events, ports and parts of the model. The variables of the selected instance can be seen and the variables, defining primitive data types can be edited during the simulation.

Topology /sequence /activity

Communication

# 3.4 Enterprise Architect

Enterprise Architect is a product of Sparx Systems is a modelling, design and management platform based on UML and related stndarts. The evaluated version is 10.0 with build version is 1006, the program is used in Ultimate edition.

In Professional and above editions Enterprise Architect can simulate Activity diagrams, State machines and Sequence diagrams, using a simulation engine, therefore no code will be generated for the simulation purposes. In Corporate and above editions, Javascript can be used as a run-time execution language in order to define expressions and behaviours. However the ability to use Javascript is only enabled for State machines and Activity diagrams, in the evaluated version Javascript can not be used in Sequence diagrams.

3.4.1 Execution & Debugging

Given that, Javascript can not be used in all editions and Diagrams there are two kinds of execution types. : manual and interpreted. In manual execution all decisions are made manually and no Javascript will be interpreted. In interpreted simulation the models can be executed dynamically using Javascript. In the further part we will be talking about this kind of simulation.

All diagram types, that are supported in simulation, can be executed without having an owning class. Because the communication is not established over classifiers, this does not have a negative effect on simulation. A global variable which is not owned by a class can be defined using the prefix “sim.”. This kind of variables can be accessed global in the simulation and must not be declared as an attribute of a class or component.

An attribute of a class or component can be accessed using the prefix “this.”. In both cases the variables must be initialized manually in the model or in the Simulation script .The variables defined with prefixes “sim” and “this” will be shown in “locals” window. Other variables can also be defines without these prefixes, but those wont be dipslayed in “locals” windowand are not traceable. All threads have access on Local variables. If any thread specific variable is desired these must be initalized in the specific thread.

A simulation can be started using the command “Execute/Interpreted or Manual”, by using a Simulation script or by clicking the “run” button in simulation window for the active diagram. Simulation scripts enables the user to set up the simulation type, entry point(the diagram to execute) and the Platform type(UML or BPMN). Additionaly the simulation variables can be initalized using the “input” field which will execute prior to the simulation being run.

Enterprise architect can run only one simulation session at once. Once a simulation session is started, using “Simulation” window, the simulation can be controlled with “run”, “pause”, “step over/in/out” and ”stop” buttons. The simulation speed can be set up, also during the simulation, using the “speed” field. During execution the “simulation” window displays the currently executing step, system outputs and other informations such as warnings or errors. System outputs can be defined using Javascripts “Trace()” method. The run-time values of the primitive variables can be changed using Console input area, under the simulation toolbar.

Breakpoints window allows users to define various breakpoint sets. Breakpoints can be set to various Uml elements such as states, actions, transitions, decisions etc.. .Elements does not have a direct command to add a breakpoint in their context menus, in order to do that, first the element must be selected in diagram and then “add breakpoint” button must be clicked in breakpoints window.

Furthermore in Events window, the execution of the events can be controlled. In the evaluated version, only signal events are supported in simulation. Using this window, users can create custom trigger sets and can save the a sequence of triggers from earlier simulations as sets for reuse. Any fired, lost and waiting triggers will be listed during the simulation. When the simulation arrives a point, where a trigger is needed to proceed the execution, the available triggers that can be fired are listed in the “ waiting triggers” part of the window. To fire a trigger that is not in the waiting triggers list, the user should drag and drop the trigger from project browser into events window. Signal events can also have attributes.

For evaluating multi-threading models, Call stack window displays the active threads of the simulation and their current executing elements. Threads can be generated using fork nodes in activity diagrams and orthohonal states in state machines, more about this will be discussed in specific topics. Furthermore in Record&Analyse window all activities in the simulation will be recorded and can be saved for further analyse. Using the records, Sequence diagrams can be generated which shows the interaction between Behavioural Elements such as Statemachines, states and activities but not the interaction between instances of classifiers.

3.4.2 Communication between Diagrams

Enterprise Architect does not use classifiers to establish a communication between Diagrams. There are two alternatives to raise a signal: using Javascript command “BroadcastSignal([Signal name])” and using predefined actions such as Send signal and Broadcast Signal of Activity diagrams. All alternatives uses the same method of sending signals. Since there is no possibility to define a target element to send a signal, the signal names must be unique in the current simulation. In case that there are two triggers which are fired by the same signal event, two diffarent triggers with the same name will be generated by Enterprise Architect.????

The sent Signals can fire the triggers which are defined on Transitions in Statemachines and Accept Event Action whichs are defined in Activity diagrams.

Statemachines can communicate with each other using BroadcastSignal or calling activities that sends the appropiate Signal.

In the case that two or more state machines are executed parallel, they can communicate with each other. To realize this case, each of these state machines must be initially started from some sort of behaviour. We can do this using parallel running call behaviour actions or submachine states.

On the other hand if a statemachine calls/executes an another statemachine, only the callee state machine can send a signal to its caller state machine.

The same semantic for sending signals is also implemented by Activity diagrams.

Since teh dynamic simulation and use of javascript is not supported for Sequence diagrams, they dont have the capability of communicating with other diagrams during the simulation.

3.4.3 State Machines

As mentioned before Enterprise Architect uses a simulation engine to execute state machines. The core features in the simulation includes using Javascript to define actions and guards, simulating parallel flows, manually or via script firing of Triggers.

Various elements such as orthogonal/ submachine/composite states, Transitions, Choice/Junction points and Inital/final states are supported in the simulation.The list of all supported elements can be seen on xxx .History states are not supported in the simulation.

Multiple entry, do and exit actions can be defined for a state but only one of each will be executed in simulation. State actions can be defined using action code with javascript or selecting an executable behaviour instead. Using orthogonal states two or more concurrent regions can be simulated. It must be pointed out that, if any transition is triggered that exits it`s source state`s parent region, all other concurrent regions are instantly exited, irrespective of their current state.

Transitions can be triggered using Signal events, Time/Call/Change events are not supported in the simulation. Transitions can have guards, however guards can only be interpreted in dynamic simulation mode. Just like the state actions, Transitions actions can also define a behaviour using code or an executable behaviour.

Given that, the Signal names must be unique in the whole simulation, some cases can not be decided by the semantic of the execution but by the user. We consider the case that two transitions are in conflict with each other(having the same signal as trigger) in a nested state, and the active state is the inner one. The decision must be made by the user since ,as mentioned in Communication topic, Enterprise architect generates two triggers, defined by the same name.

State machines can execute other state machines or activities using state actions or transition effects without initalizing any classifier. An other way to execute a behaviour is instancing a classifier to a state. In this case the classifier is not a Class but a behaviour such as a state machine or an activity. Such a classifier can be instanced to a state with command “Instance Classifier” under the point “advanced”in the context menu of the state.

An another statemachine can be displayed and simulated in the current statemachine, although it is not a part of the current state machine which provides an ease of simulating multiple statemachines in one window.

3.4.3 Activity Diagrams

3.4.5 Sequence Diagrams

3.4.5 Validation

3.4.5 Mock Up & Gui

[10]<http://www.sparxsystems.com/enterprise_architect_user_guide/9.3/model_simulation/model_simulation.html>