Java-Simulator für Aktivitätsdiagramme

Zugehöriges Metamodell: activities_mm_flow.mdzip
Zugehöriges Beispielmodell: Act_DataFlow.mdzip

Eclipse-workspace-Name: ws_Activity_Diagramm_5

HowTo: Ein Aktivitätsdiagramm in MagicDraw erstellen oder das bestehende Diagramm ändern. Der oAW-Hauptworkflow heißt AD_main.oaw. Im dump.xmi wird das instanzierte Ecore-Modell abgebildet. Die erstellten Java-Zieldateien sind im Ordner src-gen zu finden. Um diese Dateien auszuführen, müssen sie in ein Java-Projekt (z.B. Act-Environment) kopiert/verschoben werden. Die Main-Klasse und main-Methode befindet sich in MainGUI.java. Falls das Metamodell neu übersetzt werden soll, muss der oAW-Workflow wf-uml2ecore-statem.oaw ausgeführt werden.

Tool-Versionen:

Das Projekt wurde mit eclipse3.4 (Ganymede) erstellt und die benutzte oAW-Version ist 4.3.1. MagicDraw war in der Version 16.5 SP2

Features:

- Fork/Join für Kontrollflüsse
- Decision/Merge für Kontrollflüsse
- TimeEvent mit relativer Zeit. Es werden nur ganze Sekundenwerte unterstützt.
- SendSignalAction und AcceptEventAction, sensitiv auf ReceiveSignalEvent. Der ReceiveSignalEvent muss ein Signal als Trigger besitzen. Jedes Signal muss von mindestens einem SSA getriggert werden.
- Genau einen Start-Knoten und n ActivityFinal-Knoten, die den Simulator sofort stoppen (Die JVM killt)
- CallBehaviorActions mit mehreren Kontrollflüssen als Eingang und Ausgang
- CallBehaviorActions mit Pins als Eingang und Ausgang. Diese Pins müssen mit einen Objektfluss verbunden werden, der auf der anderen Seite entweder mit einem Datenobjekt, oder mit einem anderen Pin einer CallBehaviorAction verknüpft ist.
- Eine CallBehaviorAction kann 0…n Eingangs- und 0…m Ausgangspins haben.
- Datenobjekte und Pins müssen einen primitiven Typ haben (int, boolean, double)
- Die Typen am Anfang und am Ende eines Objektflusses müssen gleich sein
- Jeder Knoten wird in einem eigenen Thread ausgeführt, es können also mehrere Knoten zur selben Zeit aktiv sein
- Hat ein Knoten (Fork, CallBehaviorAction) mehrere Eingangskontrollflüsse, werden diese synchronisiert, d.h. erst wenn alle eingehenden Transitionen geschaltet wurden, wird der Knoten aktiv (gemäß der UML-Semantik) und schaltet die Ausgangstransition(en)
- CBAs mit verschiedenen Semantiken, abhängig vom Namen bzw. Namensteilen. Folgende Namen und Funktionen sind zugeordnet:
    - Add oder add -> Die Datenwerte an den Eingangspins werden miteinander addiert und an den Ausgangspins zur Verfügung gestellt
    - Mul oder mul -> Gleiche Funktion wie Add, aber mit Multiplikation
    - Generator oder generator -> Wenn noch eine Zahl im Namen enthalten ist, beispielsweise DataGenerator(5), wird diese Zahl an den Ausgangspins zur

Verfügung gestellt. Ein Datengenerator darf keine Eingangspins und somit Eingangsdatenflüsse besitzen, sondern nur eingehende Kontrollflüsse!
    - Sind die oben angeführten Namen nicht enthalten, werden standardmäßig die Werte an den Eingangspins in der Konsole ausgegeben.
- Ist kein semantisch hinterlegter Name angegeben, wird auf der Konsole angezeigt, wenn der Knoten aktiv ist und sofort dessen ausgehenden Transitionen gefeuert.
- Einfache GUI zur Kontrolle der Simualtion und durch externen Triggerung von Signalen, siehe Abschnitt Update 27.11.09

Nicht unterstützt:

- Objektflüsse, die ohne einen Pin auf eine CBA gehen
- Mischung von Pins und Kontrollflüssen für „eine Richtung“ an CBAs, d.h. entweder gehen in eine CBA Objekte über Pins ODER Kontrollflüsse. Genau dasselbe gilt für den Ausgang. Allerdings können Kontrollflüsse in CBAs hinein und Objekte hinaus gehen und umgekehrt.
- Objektflüsse in oder aus Fork/Join- und Decision/Merge-Knoten
- Ein/Ausgangs-Parameter an der Activity (Am Diagrammrahmen!)
- Pins und Objektflüsse an SendSignalActions und AcceptEventActions (weder für ReceiveSignalEvents oder TimeEvents)
- FlowFinal-Knoten (Nur ActivityFinal-Knoten, diese terminieren die Ausführung sofort)
- Erweiterte Features von DataStore- oder CentralBuffer-Knoten
- Gewichtete Kanten
- Unterscheidung zwischen FIFO, LIFO, unordered
- Speicherung von Token und Prüfung, ob ein Folgeknoten feuerbereit ist
- Strukturierende Elemente wie
    - Zusammengesetzte Actions
    - Expansion Region
    - Interruptible Regions und Exception Handler
    - Conditional Nodes (If, Loop, Sequence)
- Unterscheidung zwischen InputPin, Value Pin und Action Input Pin
- Unterscheidung zwischen CallOperationAction, Opaque Action und Any Action
- Speicherung von Daten/Kontroll-Token. Wird ein CBA mit einem neuen Wert angestoßen, obwohl er den alten Wert noch nicht verarbeitet hat (wenn z.B. die CBA auf einen Wert an einem anderen Eingang wartet), so wird der alte Wert verworfen/überschrieben
- Pre- und Postconditions

Update 17.8.2009: Das Multithreading wurde überarbeitet:

- Bisher wurde zum Programmstart für jeden Knoten ein eigener Thread generiert. In der run-Methode eines Threads war bisher eine while(true)-Endlosschleife, in der kontinuierlich alle 100ms eine stop-Variable abgefragt wurde. Wenn die Variable „true“ war, legte sich der Thread erneut für 100ms schlafen. Die stop-Variable wurde auf „false“ gesetzt, wenn die Activity gefeuert wurde.
- Dieser Mechanismus wurde überarbeitet. Nun bekommt jede Activity die Instanz einer Semaphore-Klasse. Der Wert der Semaphore beträgt initial 0. Die run-Methode einer Aktivity ruft nun beim Eintritt als erstes ein lock() auf die Semaphore auf. Da der Wert bei 0 ist, wird der Thread in ein wait()-Statement geleitet und blockiert. Erst wenn die Aktivity feuert, wird in der executeAction()-Methode ein unlock() auf die Semaphore aufgerufen. Dies wiederum weckt den blockierten Thread und dieser kann

laufen. Im Anschluss an einen Durchlauf wird in der run()-Methode erneut ein lock() aufgerufen und der Thread blockiert wieder. Es ist wiederum ein feuern der Activity notwendig, um den Thread wieder zu aktivieren.
- Mit dieser Verbesserung wird das kontinuierliche und oft nicht erfolgreiche Pollen der Threads auf das aktivierende Ereignis entfernt.
- Falls ein Knoten aktiv ist und während dessen erneut per Kontroll-Token gefeuert werden soll, wird seine Semaphore erhöht. Wenn der Knoten dann abgearbeitet ist, versucht er seine Semaphore zu sperren. In diesem Fall ist das erfolgreich und der Knoten wird erneut aktiv. Somit werden Kontroll-Token an eingehenden Kanten gesammelt gemäß UML-Spezifikation.

Update 23.11.09
- Für jeden Knoten wird eine eigene Java-Klasse erstellt, damit können die speziellen Eigenschaften für jeden Knoten direkt in seiner Klasse abgebildet werden. Dies ist mit der der Benutzung eines Codegenerators relativ einfach. Generelle Eigenschaften von Knoten sind in einer abstrakten Basisklasse *ActivityNode* hinterlegt. Dies Klasse erbt das Interface Runnable und stellt eine Standard run()-Mothode zur Verfügung.
- Für Transitionen gibt es eine abstrakte Klasse Transition und die abgeleitete Klassen ControlFlow und DataFlow, die für jede Transition instanziiert werden.
- Erweiterte Features zum Multithreading werden aktuell noch nicht unterstützt, z.B.
    - Nebenläufige Threads können nicht miteinander kommunizieren (keine Deltazyklen)
    - Der synchronisierte Zugriff nebenläufiger Threads auf gemeinsame Ressourcen ist ebenfalls nicht möglich
- Ablaufsynchronisierung kann durch Join-Knoten erfolgen oder durch die implizite Synchronisierung wenn mehrere Kontrollflüsse in eine CallBehaviorAction eingehen

Update 27.11.09
- Das Programm ist nun GUI-gestützt:
    - Im vollständig automatisch erzeugten Code ist eine GUI hinterlegt. Diese startet, wenn die main-Methode der Klasse MainGUI ausgeführt wird.
    - Die Stadardkonsole wird auf eine TextArea umgeleitet. So kann der Simulationsfortschritt beobachtet werden
    - Die GUI bietet drei Simulationskontrollbuttons: „Start“, „Ende“ und „Freeze/Release Output“, Mit letzterem kann die Textausgabe eingefroren und wieder aktiviert werden. Dies ist insbesondere nützlich, wenn Abläufe mit Schleifen simuliert werden
    - Sind im Modell SendSignalActions ohne eingehende Kontrollflüsse modelliert, werden auf der im unteren Bereich der GUI Buttons für die SSAs angelegt. So können auch hier externe Signale vom Benutzer manuell getriggert werden.

Vorteile:
- Simulation von Kontroll- und Datenflluss, modelliert mit UML-Aktivitätsdiagrammen
- Modell basiert auf Metamodell
- Die wichtigsten Knotentypen des UML-Aktivitätsdiagramm werden unterstützt
- Jeder Knoten wird in einem eigenen Thread ausgeführt. Ist ein Knoten inaktiv, ist der Thread blockiert.
- Einfache Datenverarbeitungen können mit den hinterlegten Semantiken modelliert werden.

- Wenn mehrere Kontroll- oder Datenflüsse in oder aus einem Knoten gehen, verhält sich der Simulator gemäß der UML-Semantik, d.h. es findet eine implizite Synchronisierung statt
- Kontroll-Token werden gemäß UML-Spezifikation an einer Kante gesammelt, wenn der folgende Knoten noch aktiv und nicht feuerbereit ist.
- Einfache Simulation der Umwelt durch extern triggerbare Signale

Beschränkungen:

- Keine vollständige Unterstützung von UML-Aktivitätsdiagrammen
- In den Knoten kann kein Zugriff auf (gemeinsam genutzte) Ressourcen erfolgen
- Keine Kommunikation zwischen den Knoten
- UML-Spezifikation für die Prüfung in Fork/Join-Knoten, ob ein Folgeknoten feuerbereit ist und ggf. Token gesammelt werden, ist nicht umgesetzt – Alle Ausgänge werden unmittelbar geschaltet, wenn alle Eingänge geschaltet wurden
- Keine Unterstützung von Datenflüssen in Fork/Join- und Decision/Merge-Knoten