

Business Process Simulation **in Action**

In der letzten Ausgabe des Java Magazins wurden die Grundlagen zu Business Process Simulation (BPS) gelegt und gezeigt, wie ein entsprechendes Tool auf Basis von Open-Source-Komponenten entwickelt werden kann [1]. Dieser Artikel zeigt die Konfiguration und Verwendung dieses Tools anhand eines Beispiels.

von Bernd Rücker

Um den letzten Artikel knapp zusammenzufassen: BPS ermöglicht es, für neue oder überarbeitete Prozesse sowie für veränderte Rahmenbedingungen Vorhersagen über Kosten bzw. Leistungsparameter zu treffen. Dadurch ermöglicht Simulation Prozesse besser zu verstehen, bevor sie implementiert und produktiv gesetzt werden, was vielen Risiken entgegenwirkt. Das in [2] entwickelte Open Source Tool realisiert BPS, indem das bestehende Discrete Event Simulation Framework DESMO-J [3], [4] mit der Business Process Engine JBoss jBPM [5] verheiratet wird. Vereinfacht gesagt, funktioniert dies, indem das Simulations-Framework die Prozessmaschine steuert, welche ihrerseits Events erstellt und der Simulation übergeben kann.

Zur Verdeutlichung ist in Abbildung 1 ein beispielhafter Ablauf gezeigt. Er illustriert das Abarbeiten eines Prozess-Start-Events durch die zentrale Steuereinheit in DESMO-J:

- Bevor ein Event zur Ausführung kommt, wird die Modellzeit der Simulationsuhr zu jBPM übertragen, da die Modellzeit nicht der realen Zeit entspricht, man aber trotzdem korrekte Logdaten in jBPM abfragen möchte.
- Das Event, implementiert als Java-Objekt, wird ausgeführt. Dies startet eine neue Prozessinstanz in jBPM und der Prozessgraf wird bis zum ersten Wartezustand durchlaufen.
- Beim Erreichen des Wartezustands wird ein Event erstellt und DESMO-J übergeben. Dieses Event sorgt später für die Beendigung des Wartezustands. In diesem Fall wartet man auf das Eintreffen des Pakets, also ein externes Event. Es könnte auch eine Aufgabe für einen Sachbearbeiter im Prozessablauf vorkommen, dann entspräche das erstellte Event der erfolgreichen Bearbeitung der Aufgabe.
- Die Zeit, zu der das Ereignis eintreten soll, wird durch konfigurierte statistische Verteilungsfunktionen ermittelt, da Paketlaufzeiten oder auch Bearbeitungszeiten immer ein Stück weit zufällig sind.

- Das Prozess-Start-Event plant sich daraufhin selbst für einen späteren Zeitpunkt wieder ein. Wann genau, entscheidet wiederum eine Verteilungsfunktion.

Kurzüberblick JBoss jBPM und jPDL

Die Business Process Engine jBPM enthält innerlich keine Magie, letztendlich bildet sie einen einfachen Zustandsgraphen ab. Dabei gibt es, wie in Abbildung 2 gezeigt, Nodes und Transitions, wobei Letztere zwei Nodes miteinander verbinden. Dieses Grundprinzip ermöglicht es, Prozesse als freie Grafen zu beschreiben, ist dafür aber auch bereits ausreichend. Es gibt unterschiedliche Arten von Nodes, ich möchte an dieser Stelle nur Wartezustände („state“) und Aufgaben („task-node“) gesondert erwähnen. In beiden muss die Ausführung des Prozessgraphen unterbrochen werden, im ersten Fall, weil auf ein externes Ereignis gewartet wird und im zweiten Fall, weil erst ein Bearbeiter seine Aufgabe erledigen muss. In Wartezuständen wird der Prozesszustand normalerweise persistiert. Es sei aber angemerkt, dass



Den Quellcode zu diesem Artikel finden Sie unter www.javamagazin.de

Persistenz in der Simulation nicht vonnöten ist, da die Simulation normalerweise am Stück ausgeführt wird, ohne den Rechner zwischendurch abzuschalten. Dies wirkt sich dann positiv auf die Performance aus.

Die Prozessausführung wird durch ein Token-Objekt gesteuert, das durch den Prozessgraphen „wandert“, es entspricht also einer Prozessinstanz. Sind mehrere ausgehende Transitions an einem Node vorhanden, müssen jBPM-Informationen vorliegen, die eine Entscheidung ermöglichen, ansonsten wird die Standard-Transition verwendet.

Beispielprozess und Simulationsziel

In der letzten Ausgabe wurde bereits ein Prozessbeispiel eingeführt, das als Tutorial zum entwickelten Simulations-Tool online verfügbar ist [6]. Der Prozess ist in Abbildung 3 grafisch dargestellt und schnell erklärt: Es handelt sich um die vereinfachte Abwicklung von Retouren bei einem Webshop. Der Shop prüft zurückkommende Ware in einem Schnelltest auf Funktionstüchtigkeit. Wird kein Fehler festgestellt, so wird die Ware von einem Techniker eingehend unter die Lupe genommen. Kann der Fehler nachvollzogen werden, bekommt der Kunde sein Geld zurück. Funktioniert die Ware auch nach der zweiten Prüfung problemlos, wird die Retour abgelehnt und die Ware zurückgesendet. Der Quellcode des Prozesses liegt als XML-Datei in der Sprache jPDL (jBPM Process Definition Language) vor und ist in Listing 1 abgebildet.

Erste Fragestellung an die Simulation ist die beste Personalplanung. So wird beispielsweise um die Weihnachtszeit ein höheres Retouraufkommen angenommen, welches natürlich einen stärkeren Personaleinsatz erfordert. Die Herangehensweise bei der Simulation ist nun folgende: Es werden Szenarien mit verschiedenen Personaleinsatzstrategien entwickelt und durch die Simulation „getestet“, um so das beste Szenario zu evaluieren. Welche Kennzahlen in die Bewertung des Gewinners eingehen, ist von Projekt zu Projekt verschieden, normalerweise

ist es ein Kompromiss zwischen Kosten und Durchlaufzeit.

Statistische Zahlen zu Bearbeitungs- oder Wartezeiten stehen entweder als historische Logdaten in jBPM zur Verfügung oder müssen manuell erstellt werden. In Abbildung 3 sind die durchschnittliche Bearbeitungszeit sowie die Standardabweichung bereits eingetragen.

Die Erstellung und Konfiguration eines Simulationsprojekts erfolgt für die oben genannte Fragestellung etwa in folgenden Schritten:

- Erstellung des Prozesses in jBPM, falls noch nicht vorhanden
- Erstellung einer Simulationskonfiguration als XML, welche die ermittelten statistischen Verteilungen berücksichtigt
- Definition von Szenarien mit unterschiedlichen Resource Pools und eventuellen anderen Unterschieden

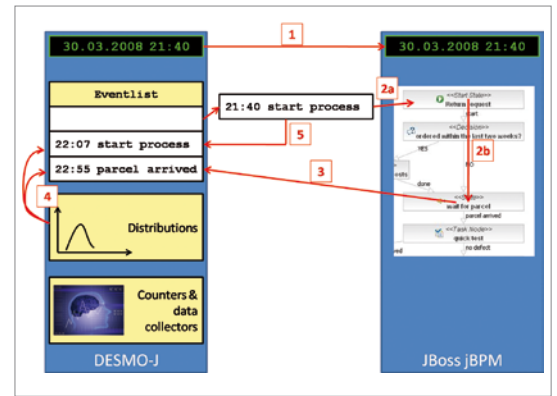


Abb. 1: Ablauf der Simulation

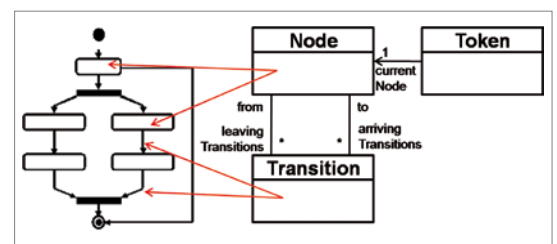


Abb. 2: Interner Aufbau von jBPM

Listing 1: jPDL-Quellcode des Beispielprozesses

```
<?xml version="1.0" encoding="UTF-8"?>
<process-definition name="ReturnDefectiveGoods">
  <swimlane name="clerk">
    <assignment pooled-actors="clerk" />
  </swimlane>
  <swimlane name="accountant">
    <assignment pooled-actors="accountant" />
  </swimlane>
  <swimlane name="tester">
    <assignment pooled-actors="tester" />
  </swimlane>
  <swimlane name="dispatcher">
    <assignment pooled-actors="dispatcher" />
  </swimlane>
  <start-state name="Return request">
    <transition name="start" to="ordered within the last
      two weeks?" />
  </start-state>
  <decision name="ordered within the last two weeks?"
    expression="#{decisionOne}">
    <transition to="transfer shipping costs" name="YES" />
    <transition to="wait for parcel" name="NO" />
  </decision>
  <task-node name="transfer shipping costs">
    <task name="transfer shipping costs"
      swimlane="accountant" />
    <transition name="done" to="wait for parcel" />
  </task-node>
  <state name="wait for parcel">
    <transition name="parcel arrived" to="quick test" />
  </state>
  <task-node name="quick test">
    <task name="quick test" swimlane="clerk" />
    <transition to="extended technical test" name="no defect" />
    <transition to="Refund" name="defect approved" />
  </task-node>
  <task-node name="extended technical test">
    <task name="extended technical test" swimlane="tester" />
    <transition to="send back goods" name="no defect" />
    <transition to="Refund" name="defect approved" />
  </task-node>
  <task-node name="send back goods">
    <task name="send back goods" swimlane="dispatcher" />
    <transition to="Denied" />
  </task-node>
  <task-node name="Refund">
    <task name="refund" swimlane="accountant" />
    <transition to="Approved" />
  </task-node>
</end-state name="Approved" />
</end-state name="Denied" />
</process-definition>
```

- Definition von Datenquellen und -filtern (in diesem Artikel wird dies nicht benötigt, siehe [6] für ein Beispiel)
- Überprüfung aller Serviceaufrufe im Prozess und eventuelles Abschalten oder Mocking

Konfigurationsparameter

Für die Simulation werden zusätzliche Informationen benötigt. Bereits angesprochen wurden Verteilungen für Bearbeitungs- und Wartezeiten. Ein weiteres wichtiges Element sind Resource Pools. Diese geben die verfügbare Anzahl verschiedener Ressourcen an, jede Ressource entspricht dann einem Pool. Wird während der Simulation eine Ressource beispielsweise zur Abarbeitung einer Aufgabe benötigt, so wird diese Ressource für die benötigte Zeit aus dem Pool entfernt. Ist keine Ressource

im Pool verfügbar, so muss der Prozess auf sie warten. Dies entspricht der Realität: Hat niemand zur Bearbeitung einer Aufgabe Zeit, so muss der Prozess warten. Für Ressourcen wird dabei auch definiert, welche Kosten sie verursachen (Costs-Per-Time-Unit) und wie viel Prozent der Kosten in den Zeiten anfallen, in denen eine Ressource nicht ausgelastet ist (Unutilized-Time-Cost-Factor).

Interessant sind vor allem noch Datenquellen und Datenfilter, wie in Listing 2 gezeigt. Datenquellen können neue Daten in den simulierten Prozess pumpen und Datenfilter können Daten verändern. Wird beispielsweise der Retourenprozess gestartet, so könnte eine Datenquelle Informationen zur Retoure als Prozessvariablen bereitstellen. Ein Datenfilter könnte dagegen das Verändern der Fehlerbeschreibung

durch einen Bearbeiter in einer Aufgabe simulieren.

Wie in [1] beschrieben, will man in der Simulation oft nicht alle Serviceaufrufe normal durchführen, man möchte sie manchmal ignorieren oder durch einen Aufruf eines Mocks ersetzen. Das jBPM-Simulations-Tool ermöglicht dies durch einige Konfigurationsparameter (Listing 3).

Konfiguration der Simulation

Die Konfiguration der Simulation kann grundsätzlich auf zwei Arten geschehen. Erstens kann man die benötigten Parameter direkt im jPDL-Quellcode des Prozesses einbetten. Dies ermöglicht es, einfach und schnell bestehende Prozesse zu simulieren. Die zweite Möglichkeit ist die Verwendung einer eigenen XML-Konfigurationsdatei. Dies hat den Vorteil, dass der Prozess

Listing 2: Konfiguration von Datenquellen und Filtern

```
<scenario name="test" execute="true">
...
<variable-source name="orders"
  handler="org.jbpm.sim.tutorial.TutorialProcessVariableSource" />
<variable-filter name="orders"
  handler="org.jbpm.sim.tutorial.TutorialProcessVariableFilter" />

<sim-process path="...datasource/processdefinition.xml">
<process-overwrite start-distribution="start">
  <use-variable-source name="orders" />
</process-overwrite>
<task-overwrite task-name="change return order"
  time-distribution="change return order">
  <use-variable-filter name="orders" />
</task-overwrite>
...
</sim-process>
</scenario>
```

Listing 3: Konfiguration von Serviceaufrufen im Prozess

```
<node name="normal actions">
<transition to="special simulation actions">
<!-- This one is not executed because of default: -->
<action name="action 1" class="...TestAction" />
<action name="action 2" class="...TestAction"
  simulation='skip' />
<action name="action 3" class="...TestAction"
  simulation='execute' />

<script name='not executed in sim 1'>
  <expression>...</expression>
</script>
<script name='not executed in sim 2' simulation='skip'>
  <expression>...</expression>
</script>
<script name='executed in sim 1' simulation='execute'>
  <expression>...</expression>
</script>

</transition>
</node>
<node name="special simulation actions">
<transition to="task">
<action name="sim-action 1" class="...TestAction"
  simulation-class='...SimTestAction' />
<simulation-action name='sim-action 2' class='...SimTestAction' />

<script name='simulation script 1'>
  <expression>...</expression>
  <simulation-expression>...</simulation-expression>
</script>
<simulation-script name='simulation script 2'>
  <expression>...</expression>
</simulation-script>
</transition>
</node>
```

selbst nicht verändert werden muss und dass in einer Simulation verschiedene Prozesse angesprochen werden können. Auch ermöglicht es die eigene Konfiguration, unterschiedliche Szenarien für den gleichen Prozess zu definieren. Dies wird im Beispiel für die unterschiedlichen Personaleinsatzstrategien benötigt. Daher ist meist die Lösung mit einer eigenen Konfiguration vorzuziehen. Es ist jedoch auch eine Mischform zulässig, wobei die Einstellungen im Prozess bei Bedarf überschrieben werden. Die Konfiguration für unseren Simulations-Use-Case ist in Listing 4 zu sehen. Es wurde ein Experiment mit drei Szenarien definiert:

- *christmas*
- *christmas_normal_case*
- *christmas_worst_case*

Ein Experiment kann beliebig viele Szenarien enthalten und definiert einige globale Parameter (wie die Laufzeit über das Attribut *run-time*). Die Simulation aller Szenarien des Experiments wird dabei immer als eine Einheit ausgeführt. Bei den Szenarien ist zu sehen, dass sie nicht zwangsläufig ausgeführt werden müssen (*execute=false*) und von Basisszenarien erben können (*base-scenario=christmas*). Dies ermöglicht eine einfache Vererbung ähnlich abstrakter Klassen in Java, ist jedoch momentan auf eine Hierarchieebene beschränkt.

Verteilungen und Resource Pools müssen einerseits definiert werden, wie in Listing 4 zu sehen ist (Element

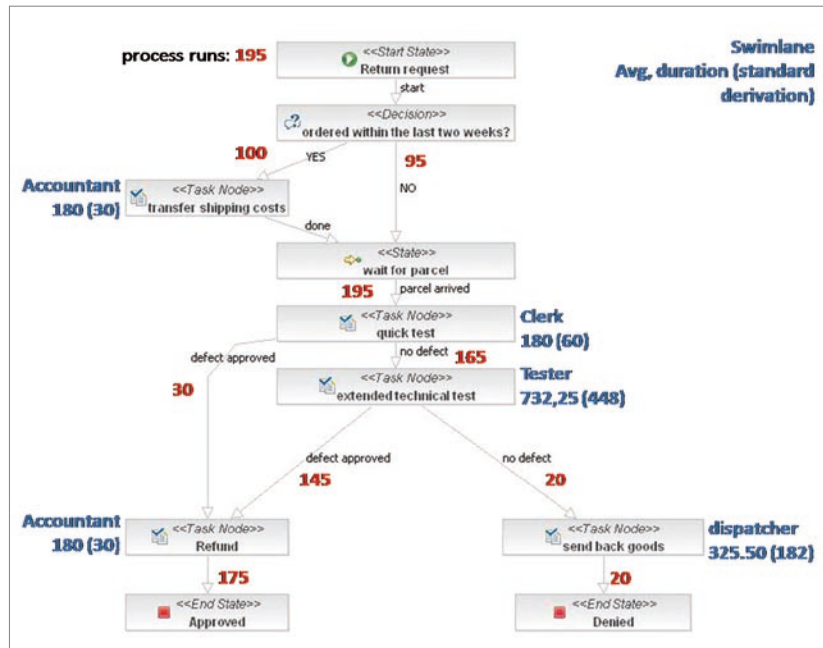


Abb. 3: Der Beispielprozess inklusive statistischer Kennzahlen

Generierung der Simulationskonfiguration aus Logdaten

Die Daten konnten in dem entwickelten Tool recht einfach aus den Logdateien gelesen werden, hierfür steht ein jBPM-Befehl als Command-Objekt zur Verfügung (das Command Pattern in jBPM abstrahiert Logik von der Umgebung, in der sie aufgerufen wird [7]. In diesem Fall wird die Umgebung als *jbpmContext*-Objekt erzeugt und übergeben):

```
BamSimulationProposal proposal = (BamSimulationProposal)
new GetSimulationInputCommand(processName).execute(jbpmContext);
proposal.createScenarioConfigurationXml();
```

Dieses Command erstellt einen Vorschlag für eine Simulationskonfiguration, die bei statischen Eingangsdaten Werte aus der Historie erschließt. Dabei muss man in der aktuellen Version noch aufpassen, da lediglich Standardverteilungen verwendet werden, was nicht in jedem Fall die beste Lösung darstellt. Daher kann es nicht schaden, mit einem Statistikprogramm wie GSTAT2 [8] zu arbeiten, um aus historischen Daten Verteilungsfunktionen zu ermitteln..



distribution), andererseits muss die Verwendung dieser Elemente konfiguriert werden. Dies wird erreicht, indem man die benannten Verteilungen dem Prozess selbst (für die Zeitabstände zwischen Prozessstarts), Wartezuständen oder Aufgaben zuweist (z.B. `<task-overwrite time-distribution=xyz>`). Resource Pools könnten auf die gleiche Art und Weise konfiguriert werden, im Beispiel wird jedoch von einer Standardeinstellung Gebrauch gemacht: Ist eine Aufgabe einer Swimlane (entspricht normalerweise einer Rolle) zu-

geordnet und auch ein Resource Pool gleichen Namens konfiguriert, so wird dieser Pool automatisch verwendet. Im Beispiel unterscheiden sich die zwei ausgeführten Szenarien übrigens nur durch die Konfiguration der Resource Pools.

Ebenfalls zu sehen ist, dass für ausgehende Transitions eine Wahrscheinlichkeit konfiguriert werden kann. Anhand dieser wird in der späteren Simulation entschieden, welche Transition zu nehmen ist. Ist keine Wahrscheinlichkeit konfiguriert, wird sich jBPM normal

verhalten (also beispielsweise die Standard-Transition verwenden).

Ausführung der Simulation

Hat man die Konfiguration erstellt, muss die Simulation ausgeführt werden. Bei der Ausführung werden statistische Daten gesammelt, die danach entweder als Java-Objekte zur automatisierten Auswertung bereitstehen oder als Reports grafisch angezeigt werden können. Letztere sind mit JasperReports [9] umgesetzt, allerdings eher als Beispiele zu sehen, da sie noch nicht sehr ausgeklügelt

Listing 4: Konfiguration der Simulation als eigenes XML

```
<?xml version="1.0" encoding="UTF-8"?>
<experiment name="ReturnDefectiveGoods"
  time-unit="second"
  run-time="28800"
  real-start-time="30.03.1980 00:00:00:000"
  currency="EUR"
  unutilized-time-cost-factor="0.0">
<!-- 28800 seconds = 8 hours = 1 working day -->
<!-- real start time only used for jbpmm audit reports -->

<scenario name="christmas" execute="false">
<distribution name="ReturnDefectiveGoods.start"
  sample-type="real" type="erlang" mean="55"/>

<distribution name="ReturnDefectiveGoods.wait for parcel"
  sample-type="real" type="normal" mean="28" standardDeviation="17"/>
<distribution name="ReturnDefectiveGoods.transfer shipping costs"
  sample-type="real" type="normal" mean="180" standardDeviation="30"/>
<distribution name="ReturnDefectiveGoods.quick test"
  sample-type="real" type="normal" mean="180" standardDeviation="60"/>
<distribution name="ReturnDefectiveGoods.extended technical test"
  sample-type="real" type="normal" mean="732.2485" standardDeviation="448.1038"/>
<distribution name="ReturnDefectiveGoods.send back goods"
  sample-type="real" type="normal" mean="325.5" standardDeviation="182.0718"/>
<distribution name="ReturnDefectiveGoods.refund"
  sample-type="real" type="normal" mean="180" standardDeviation="30"/>

<sim-process path="/ReturnDefectiveGoods/processdefinition.xml">
<process-overwrite start-distribution="ReturnDefectiveGoods.start"/>
<state-overwrite state-name="wait for parcel"
  time-distribution="ReturnDefectiveGoods.wait for parcel">
  <transition name="parcel arrived" probability="195"/>
</state-overwrite>
<decision-overwrite decision-name="ordered within the last two weeks?">
  <transition name="YES" probability="100"/>
  <transition name="NO" probability="95"/>
</decision-overwrite>
<decision-overwrite decision-name="ordered within the last two weeks?">
  <transition name="YES" probability="100"/>
  <transition name="NO" probability="95"/>
</decision-overwrite>

  <task-overwrite task-name="transfer shipping costs"
    time-distribution="ReturnDefectiveGoods.transfer shipping costs">
    <transition name="done" probability="100"/>
  </task-overwrite>
  <task-overwrite task-name="quick test"
    time-distribution="ReturnDefectiveGoods.quick test">
    <transition name="no defect" probability="165"/>
    <transition name="defect approved" probability="30"/>
  </task-overwrite>
  <task-overwrite task-name="extended technical test"
    time-distribution="ReturnDefectiveGoods.extended technical test">
    <transition name="no defect" probability="20"/>
    <transition name="defect approved" probability="145"/>
  </task-overwrite>
  <task-overwrite task-name="send back goods"
    time-distribution="ReturnDefectiveGoods.send back goods">
    <transition probability="20"/>
  </task-overwrite>
  <task-overwrite task-name="refund"
    time-distribution="ReturnDefectiveGoods.refund">
    <transition probability="175"/>
  </task-overwrite>
</sim-process>
</scenario>

<scenario name="christmas_normal_case" execute="true" base-scenario="christmas">
<resource-pool name="accountant" pool-size="5" costs-per-time-unit="0.022222222"/>
<resource-pool name="clerk" pool-size="4" costs-per-time-unit="0.011111111"/>
<resource-pool name="tester" pool-size="11" costs-per-time-unit="0.025"/>
<resource-pool name="dispatcher" pool-size="1" costs-per-time-unit="0.013888889"/>
</scenario>

<scenario name="christmas_worst_case" execute="true" base-scenario="christmas">
<resource-pool name="accountant" pool-size="6" costs-per-time-unit="0.022222222"/>
<resource-pool name="clerk" pool-size="5" costs-per-time-unit="0.011111111"/>
<resource-pool name="tester" pool-size="18" costs-per-time-unit="0.025"/>
<resource-pool name="dispatcher" pool-size="1" costs-per-time-unit="0.013888889"/>
</scenario>
</experiment>
```

sind. Momentan muss die Ausführung der Simulation über ein Stückchen Java-Code gestartet werden:

```
new JbpmSimulationExperimentRunner().run(
    "/org/jbpm/sim/tutorial/business/
    simulationExperiment.xml");
```

Eine Möglichkeit, die Experimentkonfiguration direkt per Eclipse-Run as zu starten, ist allerdings im Rahmen des jBPM-Plug-ins geplant.

Abbildung 4 und 5 zeigen beispielhaft Diagramme aus dem grafischen Report. Erstere zeigt die Verteilung der Wartezeit auf freiwerdende Ressourcen (in diesem Fall der „Tester“ vor der Aufgabe Quick Test). Die zweite Abbildung zeigt dagegen die Auslastung der Resource-„Tester“ für die verschiedenen Szenarien. Aus Platzgründen ist für das gesamte Simulationsergebnis auf [2] oder [6] verwiesen, am besten lädt man sich das Tutorial herunter und spielt selbst ein bisschen damit. Es enthält auch noch einen weiteren Simulations-Use-Case, in dem Prozessalternativen evaluiert werden.

Zusammenfassung

Das gezeigte Tool ist im Rahmen einer Master Thesis entstanden und nun Teil von jBPM. Es erfüllt die grundlegenden Anforderungen an ein Business Process Simulation Tool und zeigt, dass BPS auch mit einfachen Mitteln und ohne teure Tools umgesetzt werden kann. Trotzdem sind BPS-Projekte leider noch kein Kinderspiel. Prozessoptimierung wird noch nicht unterstützt. Prototypisch wurde es mit einem simplen Algorithmus aber bereits umgesetzt. Wünschenswert wäre hier eine bessere Unterstützung durch das Tool und die Verwendung von leistungsfähigen Algorithmen, beispielsweise könnten genetische Algorithmen zum Einsatz kommen. Für dieses Thema soll eventuell eine zweite Masterarbeit ausgeschrieben werden.

Offiziell wird das Tool mit der nächsten jBPM-Version (3.2.3) released. Momentan werden noch nicht alle Features von jBPM unterstützt, es fehlen beispielsweise Subprozesse, Super-States und Timer. Auch gibt es noch kein GUI. Der Fokus lag in der ersten Version ein-

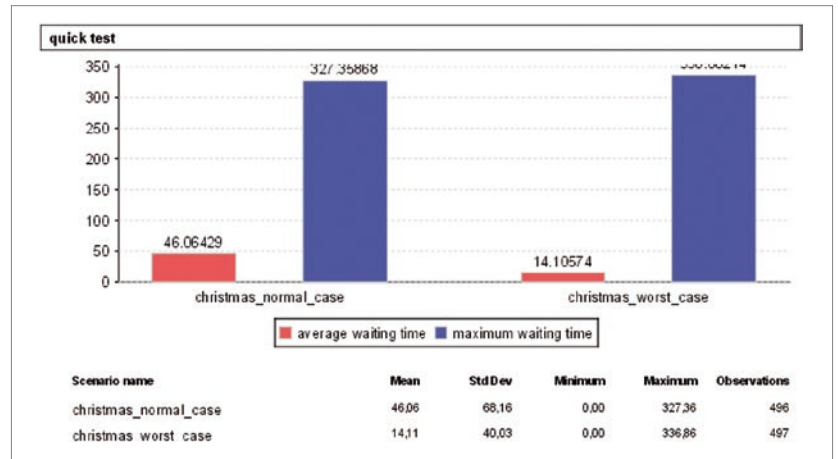


Abb. 4: Wartezeit vor Quick Test als Simulationsergebnis

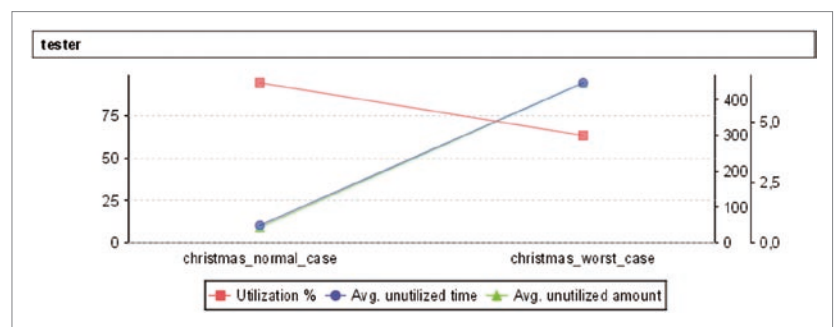


Abb. 5: Auslastung des Testers als Simulationsergebnis

deutig darauf, einen ersten stabilen Kern zu entwickeln, der die mathematische Theorie hinter BPS korrekt abbildet. Dieses Ziel konnte erreicht werden. Erweiterungen und Verbesserungen

werden später folgen, dann sollten auch mehr Erfahrungen mit dem Einsatz des Tools vorliegen. Feedback, Kritik oder Erfahrungsberichte sind dabei übrigens sehr willkommen!



Bernd Rucker ist Berater, Coach und Geschäftsführer bei der camunda GmbH. Sein besonderes Interesse liegt dabei im Bereich BPM und SOA sowie deren praktische Umsetzung, gerne auch mit Open Source. Er hat berufsbegleitend sein Studium als Master of Science in Software Technology erfolgreich abgeschlossen, in diesem Rahmen ist die angesprochene Master Thesis entstanden. Kontakt: bernd.ruecker@camunda.com

Links & Literatur

- [1] Bernd Rucker: Business Process Simulation selbst gemacht, in: *Java Magazin* 05.2008, S. 20–25
- [2] Master Thesis Bernd Rucker: www.camunda.com/content/publikationen/bernd-ruecker-business-process-simulation-with-jbpm.pdf
- [3] DESMO-J: www.desmoj.de
- [4] Bernd Page and Wolfgang Kreutzer: *The Java Simulation Handbook. Simulating Discrete Event Systems with UML and Java*, Shaker Verlag, 2005
- [5] JBoss jBPM: labs.jboss.com/jbossjbpm/
- [6] jBPM Simulation Tutorial: www.camunda.com/jbpm_simulation/jbpm_simulation_tutorial.html
- [7] JbpmCommands: wiki.jboss.org/wiki/Wiki.jsp?page=JbpmCommands
- [8] GSTAT2: www.statoek.wiso.uni-goettingen.de/user/fred/gstat2.htm
- [9] JasperReports: www.jasperforge.org/jaspersoft/opensource/business_intelligence/jasperreports/