Geschäftsprozesse kompiliert — Wichtige Unterstützung für die Modellierung

Thomas M. Prinz[1], Raphaël Charrondière[2] und Wolfram Amme[1]

[1] Friedrich-Schiller-Universität Jena, 07743 Jena, Deutschland
{Thomas.Prinz,Wolfram.Amme}@uni-jena.de
[2] École Normale Supérieure de Lyon, 69007 Lyon, France
Raphael.Charrondiere@ens-lyon.fr

Zusammenfassung Workflows — ausführbare (technische) Geschäftsprozesse — ähneln in ihrer Struktur und Funktionalität gerade im Umfeld der serviceorientierten Systeme einer modernen Programmiersprache. Der Kontrollfluss wird jedoch explizit mit Parallelität und Verzweigungsstrukturen als Graph modelliert. Die aus diesen Graphen entspringenden Programme sind dadurch selten strukturiert.
Für die Validierung und Übersetzung in ein interpretierbares und sicheres Übertragungsformat haben wir einen Compiler für Workflows entwickelt, der einen Modellierer direkt während der Entwicklung des Prozesses mit nützlichen Informationen unterstützt. Eine solche Information ist beispielsweise das Anzeigen aller möglichen Verklemmungen. Weiterhin kann der Compiler das Verhalten komplexer sogenannter inklusiver zusammenführender Gateways (OR-Joins) dem Nutzer erklären. Ein weiteres Novum ist die Benachrichtigung über mögliche Wettkampfsbedingungen und die damit zusammenhängende Transformation von unstrukturierten Programmen in die Concurrent Static Single Assignment Form.
In unserem Beitrag wird der Compiler *mojo* und seine verschiedenen Phasen vom eingehenden Geschäftsprozess bis hin zur Ausgabe vorgestellt. Im Zuge dessen gehen wir kurz auf die genutzten Analyse- und Transformationstechniken ein. Es handelt sich dabei um den ersten vollständigen Compiler für Geschäftsprozesse.

Schlüsselwörter: Compiler, Geschäftsprozess, Analyse, Verifikation

1 Einleitung

Große Softwareanwendungen werden heute im Hinblick auf Flexibilität, Skalierbarkeit und Übersichtlichkeit mit dem Pattern der *serviceorientierten Architekturen* (SOA) entwickelt. In einer SOA bieten Module ihre Dienste (Services) über Schnittstellen an. Solch ein Dienst steht dabei für die Durchführung einer (nach außen) abgeschlossenen Aufgabe. Eine Softwareanwendung, die auf dem SOA-Pattern basiert, nutzt eine Vielzahl solcher in sich abgeschlossener Dienste, um den Anwendungszweck, das Ziel, der Software zu erreichen. Wichtig ist also nicht mehr, *Wie* etwas getan wird, sondern *Was*. Sprich das *Geschäft*, das durch die

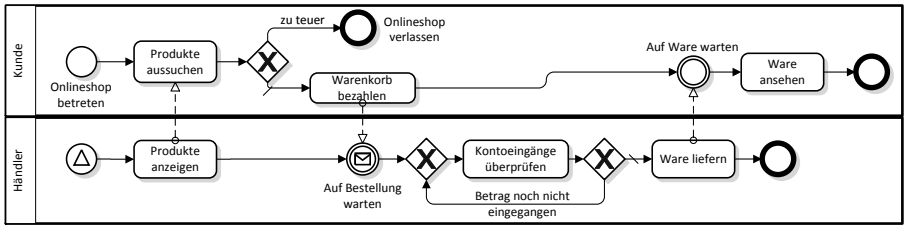

Abbildung 1. Vereinfachter Ablauf eines Einkaufs im Onlineshop

Verknüpfung mehrerer Dienste in einem Ablaufplan (*Prozess*) erledigt wird — beschrieben durch einen *Geschäftsprozess*.

Geschäftsprozesse verbinden verschiedene *Aufgaben* (Tasks) durch eine geordnete Reihenfolge (Prozess) zur Erfüllung eines Ziels (Geschäft). Abbildung 1 zeigt einen solchen Geschäftsprozess, der den (vereinfachten) Ablauf eines Einkaufs in einem Onlineshop repräsentiert und das Ziel hat, dass ein Kunde Produkte kauft, bezahlt und erhält. Die hier gewählte Notation für den Geschäftsprozess ist die *Business Process Model and Notation* (BPMN) 2.0 [1]. Der in dieser Sprache verfasste Prozess zeigt zwei involvierte Parteien: den Kunde und den Händler. Der Kunde setzt den Prozess in Gang, wenn er den Onlineshop betritt. Dies wird durch das Startevent (Kreis mit dünner Linie) gekennzeichnet. Der Teilprozess des Händlers startet ebenfalls mit dem Betreten des Onlineshops des Kunden, da er per Signal an diesen gebunden ist (Kreis mit Dreieck). Daraufhin zeigt der Händler dem Kunden seine Produkte. Daraus sucht sich der Kunde seine Produkte aus und entscheidet sich danach (Diamant mit einem X), entweder ob sie zu teuer sind oder ob er diese kauft. Sind die ausgewählten Produkte zu teuer, verlässt er den Onlineshop (Kreis mit dicker Linie) und sein Teilprozess ist beendet. Möchte er die Ware bezahlen, so geht er aus Sicht des Prozesses in die nächste Aufgabe. Ist der Kunde mit der Bezahlung fertig, so wird dem Händler eine Bestellung zugeschickt (Zwischenevent, doppelter Kreis mit Briefumschlag). Daraufhin kontrolliert der Händler den Geldeingang auf seinem Konto so lange, bis der Betrag eingegangen ist. Dann liefert er die Ware, auf die der Kunde bereits wartet. Der Teilprozess des Händlers ist damit abgeschlossen und der Kunde betrachtet abschließend noch seine erhaltene Ware.

Deutlich ist bei einem solchen Geschäftsprozess der Bezug zu Programmiersprachen zu sehen, mit zwei Unterschieden: *1)* ein Geschäftsprozess wird hauptsächlich durch eine relativ abstrakte, graphbasierte und unstrukturierte Modellierungssprache repräsentiert und *2)* besitzt einen anderen Sprachumfang.

Von diesem generellen Bezug zu Programmiersprachen ist es demnach erstrebenswert, wenn ein Geschäftsprozess ähnlich zu einem Programm in einer Programmiersprache ohne großen Aufwand automatisch *ausgeführt* werden kann. In diesem Zusammenhang wird dann über einen *Workflow*, also der technischen Umsetzung eines Geschäftsprozesses, gesprochen. Für einfache Geschäftsprozesse ist die Ausführung als Workflow derzeit sogar schon mit dem richtigen Werk-

zeug möglich (bspw. Activiti BPM Platform [2], Redhat jBPM [3] und IBM WebSphere [4]). Dabei wird der Prozess von einer *Workflowengine* interpretiert.

Ein Großteil von praxisnahen Prozessen kann jedoch nicht ohne weiteres ausgeführt bzw. als Workflow angesehen werden. Dies hat mehrere Gründe. Zum einen sind Beschreibungselemente moderner, meist graphbasierter Prozessmodellierungssprachen nicht vollständig spezifiert und deren Semantik könnte demnach unterschiedlich interpretiert werden. Dazu zählen beispielsweise die sogenannten inklusiven Gateways (OR-Split und OR-Join), die Fehlerbehandlung und Events. Zum anderen sind die existierenden Informationen im Prozess meist derart abstrakt, dass eine Übersetzung in maschinenverständliche Befehle nur durch einen Menschen vorgenommen werden kann.

Ein weiterer wichtiger Punkt ist die generelle Unterstützung des Prozessentwicklers bei der Modellierung eines Prozesses durch ein vernünftiges Werkzeug, dass dem Entwickler bereits während der Modellierung Fehler im Entwurf anzeigt. Teure Fehlverhalten sind ein wichtiger Knackpunkt zur Automatisierung. Tatsächlich gibt es bereits Werkzeuge (bspw. LoLA [5] und Woflan [6]), die in der Regel nach der Entwicklung Auskunft darüber geben, ob der Prozess frei von bestimmten Fehlern ist. Diese verwenden aber hauptsächlich Techniken zur Untersuchung des Zustandsraums, die in manchen Fällen einen exponentiell wachsenden Zustandsraum in Bezug auf die Größe des Prozesses generieren. Viel schwerwiegender ist jedoch, dass immer nur die *Fehlerwirkung*, sprich der Effekt, der durch einen Modellierungsfehler entsteht, untersucht wird. Aus Gründen der Berechnungskomplexität wird sogar nur die erste erreichbare Fehlerwirkung gesucht und danach die Fehlersuche abgebrochen. Die Information über eine solche Fehlerwirkung ist auch wichtig, kann aber nur unter Umständen zur Findung des *Fehlerzustands* bzw. der *Fehlerursache* beitragen. Dies wird in der Softwarequalitätssicherung auch als die *Entfernung* zwischen dem Fehlerzustand und der Fehlerwirkung bezeichnet. Ebenso ist auch die Entfernung zwischen der Fehlerwirkung von der eigentlichen *Fehlhandlung* (bspw. der falsche Einsatz eines Beschreibungselements) entsprechend hoch. Erschwerend kommt noch hinzu, dass sich zwei Fehler gegenseitig *maskieren* können, d.h., die Fehlerwirkung eines Fehlerzustands wird durch eine andere Fehlerwirkung eines Fehlerzustands korrigiert. Auch das *Blockieren* von weiteren Fehlerzuständen durch eine bereits aufgetretene Fehlerwirkung ist möglich — zum Beispiel bei einer Verklemmung, die alle nachfolgenden Fehlerzustände unerreichbar macht.

Abschließend zur Unterstützung bei der Modellierung finden außerdem Dateninformationen in geläufigen Techniken kaum bis keine Betrachtung und auch die Visualisierung und detailierte Beschreibung von gefundenen Fehlern (seien es Fehlerzustände oder -wirkungen) ist sehr unausgeprägt.

Insgesamt sehen wir den Bedarf der Evolution der Geschäftsprozesse, Prozessmodellierung und -ausführung hin zu einer (wenn auch abstrakteren) graphischen Programmiersprache mit einer integrierten Entwicklungsumgebung (IDE) bestehend aus einem Prozessdesigner, einem *Compiler* und der dazu passenden *virtuellen Maschine* (VM). Dafür zwingend erachten wir die folgenden Schritte: *1)* die Herleitung und Spezifikation einer relativ kompakten Kernsprache zur

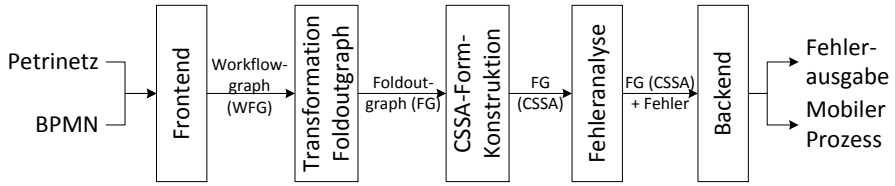

Abbildung 2. Phasen des Compilers

Prozessmodellierung mit häufig verwendeten Sprachelementen inklusive deren eindeutiger Semantik, *2)* die Herleitung und Spezifikation von Befehlen dieser Sprache zur Modifizierung von Informationen, *3)* die Definition von potentiellen Fehlerwirkungen und Fehlhandlungen, deren Fehlerursachen gefunden, angezeigt und detailliert beschrieben werden sollen, *4)* die Entwicklung von Anlaysen zur Findung von Fehlerursachen, *5)* die Entwicklung eines Compilers, der den Prozess einliest, nach Fehlerursachen und -wirkungen analysiert ggbf. Fehler anzeigt und beschreibt sowie in ein übertragbares und ausführbares Format übersetzt, und *6)* die Entwicklung einer virtuellen Maschine, die dieses Format einliest, auf seine Richtigkeit und Fehlerfreiheit überprüft und ausführt.

Ausführlichere Beschreibungen sind in einer vorangegangenen Arbeit [7] zu finden. In dieser stellten wir auch unsere Idee für den generellen Aufbau eines solchen Systems vor. Dabei wird das System in eine *Produzenten-* (der Compiler) und eine *Konsumentenseite* (die virtuelle Maschine bzw. Engine) unterteilt.

In dieser Arbeit stellen wir die bisherige Umsetzung der Produzentenseite in Form unseres Compilers für Geschäftsprozesse vor. Unter dem Namen *mojo*[1] entwickeln wir seit 2013 einen eigenständigen Compiler zur Analyse und Übersetzung von Prozessen. Er kann in existierende Prozessdesigner eingebunden werden, um deren Funktionalität zu erweitern (bisher im Activiti-Designer [2] getestet). Die Features unseres Compilers sind zum Beispiel, dass *1)* die Fehlerursachen aller potentiellen Verklemmungen innerhalb eines Prozesses gefunden werden, *2)* eine vollständige Semantik für inklusive zusammenführende Gateways (OR-Joins) genutzt wird und *3)* mögliche Wettkampfsbedingungen angezeigt werden können.

Im nächsten Abschnitt 2 untersuchen wir die unterschiedlichen Phasen des Compilers und zeigen dessen einzigartigen Funktionsumfang. Zum Schluss geben wir einen Ausblick auf zukünftige Arbeiten und Funktionalitäten in Abschnitt 3.

2 Aufbau des Compilers

Unser Compiler *mojo* orientiert sich an klassischen Phasen, in denen der Prozess immer wieder transformiert und analysiert wird. Das Phasenmodell in Abbildung 2 zeigt die derzeitigen Phasen von *mojo*.

[1] `http://sourceforge.net/projects/bpmojo/`, `http://www.bpmn-compiler.org`

Als Eingabe des Compilers dienen entweder ein in XML linearisierter BPMN-Prozess oder ein Prozess in Form der Petri Net Markup Language (PNML) [8]. Der Eingabeprozess wird dann im *Frontend* zunächst in die Zwischenrepräsentation des *erweiterten Workflowgraphen* [9] übersetzt. Erweiterte Workflowgraphen sind technische Repräsentationen des Prozesses als Graph. In der nächsten Phase (*Transformation Foldoutgraph*) wird der erweiterte Workflowgraph in den speziell für *mojo* entwickelten Foldoutgraphen übersetzt. Ein Foldoutgraph besteht in strukturierten Graphen immer nur aus einer Sequenz von hintereinander folgenden Knoten. Jeder dieser Knoten kann dabei jedoch in einen größeren Teilgraphen des Workflowgraphen entfaltet werden. Aus diesem Grund hat der Foldoutgraph auch seinen Namen erhalten. Der Gewinn dieser Transformation in den Foldoutgraph erlaubt schnelle Analysen dank einer impliziten Dominanz- und Postdominanzbeziehung zwischen den Knoten.

Nach der Konstruktion des Foldoutgraphen werden seine Instruktionen in die Concurrent Static Single Assignment Form (CSSA) [10] überführt (Phase *CSSA-Form-Konstruktion*). Im Zuge dieses Schritts muss der Compiler analysieren, welche Instruktionen parallel abgearbeitet werden und somit Wettkampfsbedingungen erzeugen können. Die gewonnene CSSA-Form der Instruktion macht es danach leichter dieses Wissen in Analysen zu verwenden. Diese Analysen werden dadurch wesentlich einfacher und effizienter.

Der eingehende Prozess befindet sich jetzt in der endgültigen Zwischenrepräsentation. Nun werden verschiedene *Fehleranalysen* durchgeführt. Aufgrund der Foldoutgraph-Struktur werden viele der Fehler bereits während der Transformation entdeckt. An dieser Stelle werden demnach noch die unstrukturierten Bestandteile des Foldoutgraphen analysiert und weitergehende Überprüfungen durch Prädikatenanalysen durchgeführt. Die gefundenen Fehler nutzt der Compiler für deren detaillierte Beschreibung zur Weitergabe an das entsprechende Modellierungswerkzeug (Phase *Backend*). Werden hingegen keine Fehler gefunden, linearisiert das Backend den Prozess und exportiert ihn. Er kann nun einfach von einer virtuellen Maschine ausgeführt werden.

Im weiteren Verlauf beleuchten wir die einzelnen Phasen etwas gründlicher.

2.1 Frontend

Wie bereits erwähnt wird im Frontend ein eingehender Prozess in einen erweiterten Workflowgraphen übersetzt. Der eingehende Prozess ist dabei entweder in der Notation eines BPMN-Prozesses notiert oder liegt als Petrinetz (PNML [8]) vor. Ausschnitt *a)* aus Abbildung 3 zeigt dies schematisch.

Für jedes Eingangsformat gibt es einen eigenen Parser, der zunächst die Syntax überprüft. Danach wird Knoten für Knoten und Kante für Kante des Eingangsprozesses in einen Workflowgraphen übersetzt [11]. Abbildung 4 zeigt die Übersetzung des Beispielprozesses aus Abbildung 1 in einen solchen Graphen.

Ein Workflowgraph ist im Wesentlichen eine allgemeinere und technischere Darstellungsform von Prozessen. Grundlegend stellt sie — ähnlich zu einem Kontrollflussgraphen — den Prozess als einen Graphen dar. Jedoch werden die Knoten je nach Funktionalität in verschiedene Typen unterteilt. Wir illustrieren

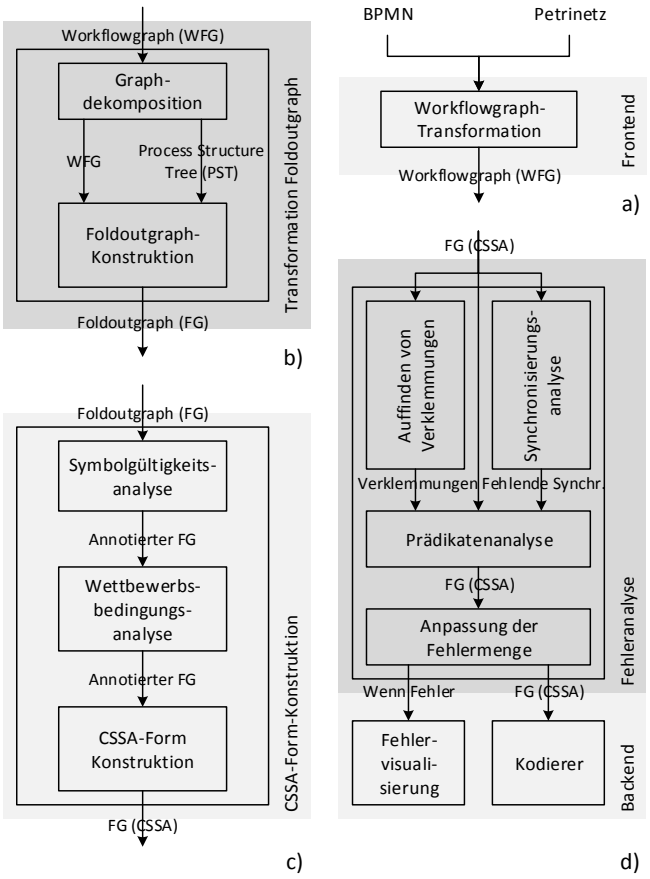

Abbildung 3. Detaillierte Phasen des Compilers

Startknoten als runde Kreise mit dünner Linie. Auf den ausgehenden Kanten dieser Knoten liegt zu Beginn der Kontrollfluss. Einfache Berechnungen, etc. werden in Aufgaben (Tasks, dargestellt als einfache Rechtecke) durchgeführt. Parallelität wird in AND-Forks erzeugt und in AND-Joins wieder zusammengefasst (schwarz-gefüllte Rechtecke). Außerdem gibt es noch XOR-Forks und XOR-Joins für die Darstellung von Verzweigungen und Schleifen, im Graphen durch Diamanten dargestellt (XOR-Forks mit dünner, XOR-Joins mit dicker Linie). Analog dazu werden OR-Forks und OR-Joins dargestellt. Sie enthalten jedoch in der Mitte des Diamanten einen schwarzen Punkt. Ein OR-Fork und OR-Join ist in der Abbildung nicht zu sehen. Zu guter Letzt gibt es noch die Endknoten (Kreise mit dicker Linie), wobei in jedem Endknoten maximal ein Kontrollfluss endet.

Eine weitere Aufgabe des Frontends ist die Vereinigung der Start- und Endknoten zu einem einzigen Start- bzw. Endknoten unter Berücksichtigung der Semantik der Eingangssprache. Dies dient vor allen Dingen der wesentlichen

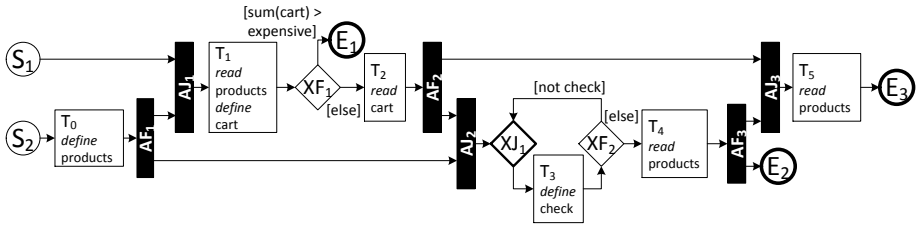

Abbildung 4. Der erweiterte Workflowgraph des Prozesses aus Abbildung 1

Abbildung 5. Verknüpfung mehrerer Start- und Endknoten zu einem einzigen

Vereinfachung von Analysen und Transformationen in den weiteren Phasen des Compilers. Vor unserer Einführung einer eindeutigen und vollständigen Semantik von OR-Joins [12], war das Verknüpfen verschiedener Endknoten zu einem einzigen uneffizient. Nun können (mit den passenden Bedingungen) die existierenden Start- und Endknoten jeweils durch einen Task ersetzt und mit Hilfe eines OR-Forks bzw. OR-Joins verknüpft werden, wie in Abbildung 5 zu sehen ist.

Außerdem transformiert das Frontend auch die Instruktionen des Prozesses. Die Erweiterung von Workflowgraphen mit Instruktionen wurde dabei erstmals in einer unserer früheren Arbeiten [9] genutzt. Sie werden *erweiterte Workflowgraphen* genannt. Im Zuge dieser Arbeit nutzen wir einfache Anweisungen, wie $define$ und *read*, um die Nutzung von Variablen anzudeuten.

2.2 Transformation Foldoutgraph

Der erweiterte Workflowgraph des Frontends wird an die nächste Phase des Compilers weitergereicht. Ab diesem Zeitpunkt sind die Analysen und Transformationen unabhängig von der gewählten Ausgangsnotation.

Der Compiler transformiert in dieser Phase den erweiterten Workflowgraphen in eine weitere Zwischenrepräsentation — den Foldoutgraphen. Foldoutgraphen zeichnen sich durch ihre Effizienz bei der Ausführung weiterer Transformationen und vor allen Dingen Analysen aus. Für strukturierte Prozesse besteht jeder Foldoutgraph aus einer Sequenz von Knoten — ohne Sprünge, Schleifen, Parallelität, etc. Dafür können bestimmte Knoten entfaltet werden. Diese Knoten haben dabei einen speziellen Knotentyp und stehen dabei beispielsweise für Verzweigungen, Schleifen und Parallelität.

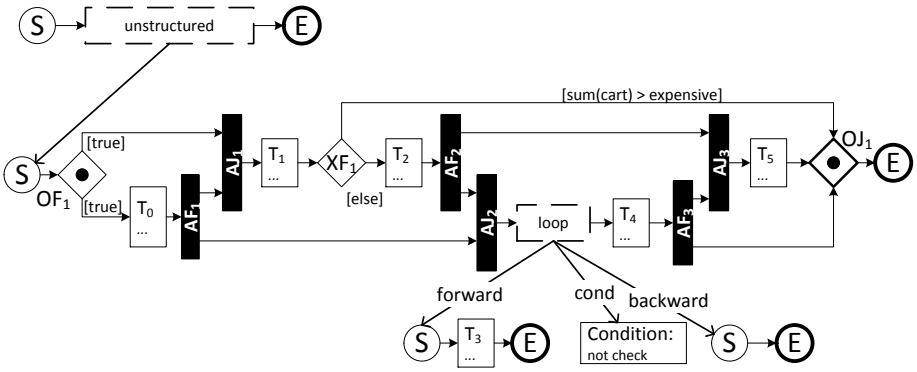

Abbildung 6. Der Foldoutgraph des Workflowgraphen aus Abbildung 4

Die Idee des Foldoutgraphen wurde von *OP2* [13], einem portablen Oberon-Compiler, und dem mobilen Code *SafeTSA* [14] inspiriert. Der Vorteil ist, dass der Dominator- und Postdominatorbaum und somit auch die Gültigkeit von Variablendefinitionen, etc. implizit gegeben sind. Außerdem ist das Auffinden vieler Modellierungsfehler schon während der Transformation möglich.

Da die meisten Workflowgraphen von realen Prozessen leider unstrukturiert sind, können Knoten des Foldoutgraphen auch zu unstrukturierten Teilgraphen entfaltet werden. Diese Teilgraphen folgen der Definition normaler erweiterter Workflowgraphen. Als Beispiel zeigt Abbildung 6 den Foldoutgraphen unseres Beispielprozesses. Da dieser stark unstrukturiert ist, sind lediglich die Schleife und das Fragment aus OR-Fork und OR-Join ausklappbar.

Die Transformation eines Workflowgraphen in einen Foldoutgraphen geschieht in zwei Schritten, wie in der schematischen Darstellung dieser Phase in Abbildung 3 *b)* zu sehen ist. Zunächst findet auf dem Workflowgraphen eine *Dekomposition* statt. Danach werden die aus der Dekomposition gewonnenen Daten genutzt, um den Foldoutgraphen zu konstruieren.

Die Dekomposition von Kontrollflussgraphen führten Johnson et al. [15] in ihrer Arbeit über den *Program Structure Tree* ein. Dabei werden *Single-Entry-Single-Exit*-Teilgraphen gesucht, wobei diese genau einen eingehenden und einen ausgehenden Knoten besitzen. Für unsere Dekomposition haben wir den Algorithmus von Vanhatalo et al. [16] übernommen, der von einem Prozess einen *Refined Process Structure Tree* (PST) ableitet. Dabei sucht er nach Regionen mit einer einzigen eingehenden und ausgehenden *Kante*.

Mit Hilfe des PST wird der Workflowgraph in linearer Zeit durchlaufen und die Fragmente des PST in Knoten und Kanten umgewandelt. Dabei wird auch bestimmt, welche Funktionalität das Fragment und somit der erzeugte Knoten widerspiegelt (Parallelität, Verzweigung, Schleife, usw.).

Während der Konstruktion des Foldoutgraphen wird außerdem überprüft, ob Fragmente korrekte öffnende und schließende Knoten haben. Beispielsweise

wird geschaut, ob ein Fragment, dass mit einem AND-Fork beginnt auch immer mit einem AND- oder OR-Join abgeschlossen wird. Diese Fehler merkt sich der Compiler, repariert aber quasi den Graphen selbst für spätere Analysen.

2.3 Konstruktion der CSSA-Form

Nach der Konstruktion des Foldoutgraphen werden seine Instruktionen in der Phase der *CSSA-Form-Konstruktion* (Abbildung 3 *c)*) in die *Concurrent Static Single Assignment* (CSSA) Form [10] übertragen. In der CSSA-Form wird jede Variable (statisch) nur genau einmal definiert und bei jeder Änderung gibt es eine Neudefinition der Variablen. Da bei der Zusammenkunft zweier exklusiver Pfade mehrere Definitionen einer Variablen vorkommen können, wird an diese Stelle eine ϕ-Funktion für jede Variable eingefügt. Diese dient als virtuelle Kopieroperation, d.h., der Wert des ausgeführten Pfades wird durch die ϕ-Funktion gewählt.

In parallelen Programmen kann außerdem eine Variable gleichzeitig an zwei unterschiedlichen Stellen im Prozess modifiziert werden. Da dadurch Analysen fehlerhafte Ergebnisse liefern, wird vor jeder dieser möglichen, gleichzeitigen Zugriffe eine π-Funktion eingefügt. Diese speichert in einer neuen Kopie der Variablen den jeweilig zuletzt gefundenen Wert.

Um die CSSA-Form ableiten zu können, muss zunächst ermittelt werden, welche Variablen überhaupt an welchen Stellen im Programm Gültigkeit haben (Schritt *Symbolgültigkeitsanalyse*). Die Gültigkeitsbereiche von Variablen in explizit parallelen, strukturierten Programmiersprachen werden von der Struktur selbst vorgegeben. Da die Gültigkeitsbereiche in Workflows bisher jedoch noch keine Festlegung haben, benutzen wir den von uns in [17] vorgeschlagenen Ansatz: *Für einen Knoten n sind alle Variablen gültig, wenn diese in einem Dominator von n ebenfalls gültig sind oder dort definiert werden.*

Dieser Ansatz folgt den strukturierten Programmiersprachen, in denen die Struktur die Dominanzbeziehung vorgibt. Innerhalb des Foldoutgraphen können somit die Variablengültigkeiten in strukturierten Bereichen durch eine Rückwärtstraversierung für einen Knoten ermittelt werden. Für unstrukturierte Bereiche wird die Dominatorrelation abgeleitet.

Bezogen auf unser Beispiel aus den Abbildungen 4 und 6 bedeutet dies, dass die Variable *products* im Task T_1 nicht gültig ist. Dies kommt daher, dass der Task T_0 den Task T_1 nicht dominiert. Entsprechend sollte die Variablendefinition nach vorn gezogen werden.

Nun kann die Transformation in die CSSA-Form durchgeführt werden. Der Algorithmus zur Erzeugung der CSSA-Form von Lee et al. [10] leitet zunächst eine partielle Ordnung des Graphen ab, um festzustellen, welche Variablenzugriffe Konflikte verursachen können. Die tatsächliche partielle Ordnung zu bestimmen, ist Co-NP-vollständig. Wie jedoch bereits Lee et al. in ihrer Arbeit berichten, wird durch eine konservative Abschätzung lediglich die Anzahl der einzufügenden π-Funktionen erhöht. Dadurch werden mehr π-Funktion eingeführt als notwendig.

Anstelle der partiellen Ordnung ermittelt *mojo* Wettkampfsbedingungen. Die Analyse von Wettkampfsbedingungen in strukturierten Fragmenten des Foldoutgraphen ist einfach: Können zwei Zugriffe auf dieselbe Variable in zwei

unterschiedlichen Knoten in Konflikt stehen, so müssen beide Knoten im Foldoutgraphen als ersten gemeinsamen Knoten eine Parallelität besitzen.

In unstrukturierten Bereichen (und somit für allgemeine Workflowgraphen) nutzen wir eine konservative Heuristik. Diese Heuristik besteht aus zwei Schritten für je zwei Zugriffe auf dieselbe Variable in zwei Knoten: *1)* Lösche von beiden Knoten die ausgehenden Kanten und verbinde beide durch ein XOR-Join (in einer Kopie des Graphen). *2)* Erreichen nun in einem Zustand zwei Kontrollflüsse das neue XOR-Join, so gibt es einen Konflikt.

Schritt *2)* der Heuristik bezieht sich auf die Fehlerwirkung einer *fehlenden Synchronisierung* — einem klassischen Fehler in der Prozessmodellierung. Wie in der folgenden Phase der Fehleranalyse beschrieben wird, haben wir einen effizienten Algorithmus entwickelt, der alle fehlenden Synchronisierungen in beliebigen Workflowgraphen findet [18].

Tests an einer Benchmark realer Geschäftsprozesse mit zufälligen Variablenzugriffen haben gezeigt, dass die Heuristik mit unserem Algorithmus zur Fehlererkennung gleiche Ergebnisse erzielt, wie die zeitaufwändige und im Zweifel exponentiell-große Zustandsraumerkundung.

Nachdem *mojo* die Wettkampfsbedingungen gefunden hat, werden diese zum einen zu deren späterer Visualisierung für den Entwickler gespeichert und zum anderen zur Platzierung der π-Funktionen genutzt. Danach werden die ϕ-Funktionen eingefügt. Die Konstruktion der CSSA-Form ist vollendet.

Wie in unserem Prozess aus Abbildung 4 gut zu sehen ist, besitzt unser Prozess keine solcher Konflikte, da die parallelen Prozesse quasi sequentiell verlaufen.

2.4 Fehleranalyse

Der ursprüngliche Prozess liegt nun in Form eines Foldoutgraphen in CSSA-Form vor. Diese (endgültige) Zwischenrepräsentation wird genutzt, um Modellierungsfehler innerhalb des Prozesses zu finden.

mojo findet verschiedene Arten von Modellierungsfehlern. Wie bereits erwähnt, analysiert der Compiler den Prozess bereits in jedem Übersetzungsschritt: *1)* syntaktische Fehler, *2)* korrekte Verwendung von öffnenden und schließenden Knoten von Fragmenten und *3)* undefinierte Variablen und Wettkampfsbedingungen.

In der Phase der *Fehleranalyse* sucht *mojo* zusätzlich nach *Verklemmungen* (Deadlocks), *fehlenden Synchronisierungen* und *nicht lebendigen Knoten*.

Verklemmungen und fehlende Synchronisierung Bei Verklemmungen und fehlenden Synchronisierungen handelt es sich um klassische Fehler während der Kontrollflussmodellierung in Geschäftsprozessen. Bei einer Verklemmung verbleibt der Prozess in ein und denselben Zustand und kann daher nicht ordnungsgemäß terminieren. Dies geschieht bei AND-Joins, die nicht auf jeder eingehenden Kante einen Kontrollfluss anliegen haben, und bei zwei OR-Joins, die sich gegenseitig blockieren. Die linke Seite von Abbildung 7 *a)* zeigt eine klassische Deadlocksituation, die durch das Zusammenführen zweier alternativer Pfade durch ein AND-Join herbeigeführt wird.

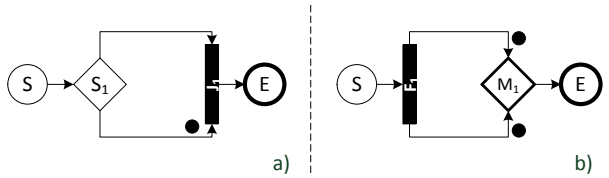

Abbildung 7. Eine Verklemmung *a)* und eine fehlende Synchronisierung *b)*

In einer Situtation mit einer fehlenden Synchronisierung besitzt eine Kante des Graphen zwei Kontrollflüsse (Tokens). Dies ist in Abbildung 7 auf der rechten Seite *b)* dargestellt. Diese Situation entsteht hier durch das Zusammenführen zweier paralleler Pfade durch ein XOR-Join.

Ursprünglich wurden die Begriffe Verklemmung (Deadlock) und fehlende Synchronisierung (Lack of Synchronization) von Sadiq und Orlowska eingeführt [19]. Seit dem entstanden die verschiedensten Techniken, um diese zu zeigen. Das Augenmerk dieser Techniken liegt auf dem Zustand, in dem die Fehlerwirkung zu Tage tritt. Die Fehler*ursache* wird nicht betrachtet. Außerdem wird nur die erste Fehlerwirkung gefunden und es wird nicht analysiert, ob nicht eine andere Fehlerwirkung bereits die Ursache für die gefundene Fehlerwirkung war.

In unseren Arbeiten zu diesem Thema [18, 20] haben wir einen neuen, compilerbasierten Ansatz entwickelt. Dieser erlaubt es bei kurzzeitiger Vernachlässigung der Dateninformationen (analog zu den bisherigen Techniken), alle *potentiellen* Fehlerursachen in beliebigen Prozessen zu finden und diese zu beschreiben. Potentielle Fehlerursachen müssen sich zur Laufzeit nicht durch eine Fehlerwirkung zeigen, da der eigentliche Kontrollfluss die Fehlerwirkung in manchen Fällen maskiert oder der Knoten, der die Fehlerursache bewirkt, gar nicht erreicht wird.

Die Algorithmen zur Analyse sind sehr effizient. Tests und Benchmarks haben gezeigt, dass dazu notwendige Analysen bereits im Hintergrund *während* der Modellierung durchgeführt werden und Ergebnisse liefern können.

Der wesentliche Ansatz, den wir verfolgen, ist das Ausfindigmachen von Knoten, deren Ausführung bestimmte Fehlerwirkungen hervorrufen können. Bspw. haben wir festgestellt, dass es auf allen Pfaden vom Startknoten zu einem AND-Join und auch von diesem AND-Join zu sich selbst Knoten geben muss, die *garantieren*, dass jede eingehende Kante des AND-Joins einen Kontrollfluss bekommt. Diese Knoten nennen wir *Aktivierungsknoten*. Die Abwesenheit eines solchen Knotens ergibt somit das Potential zu einer Verklemmung. Somit ist die Ursache einer aufgetretenen (und tatsächlichen) Verklemmung in einem AND-Join, das Fehlen eines solchen Aktivierungsknotens. Nehmen wir unser Beispiel aus Abbildung 6: Das AND-Join AJ_2 besitzt einen potentiellen Deadlock. Der Grund dafür liegt in dem XOR-Fork XF_1, das dafür sorgen kann, dass der Kontrollfluss nicht die obere eingehende Kante von AJ_2 erreicht, sondern über das OR-Join OJ_1 zum Endknoten verschwindet. Damit existiert auf keinem Pfad vom Startknoten zu AJ_2 ein Aktivierungsknoten. Bei der Untersuchung nach fehlenden Synchronisierungen verfolgen wir eine ähnliche Herangehensweise.

Die in unseren Vorarbeiten entwickelten Techniken wurden außerdem mit *mojo* weiterentwickelt, so dass auch Verklemmungen in OR-Joins erkannt werden können. Seitdem können auch Verklemmungen und fehlende Synchronisierungen erstmals in Prozessen gefunden werden, die OR-Joins beinhalten.

Prädikatenanalyse Unser Beispiel aus Abbildung 6 zeigt, dass es sich bei den gefundenen Verklemmungen und fehlenden Synchronisierungen um *konservative* Abschätzungen handelt: Wenn die Kosten für den Warenkorb ($sum(cart)$) immer kleiner oder gleich dem Wert *expensive* sind, dann tritt die gefundene Verklemmung nicht auf.

Diese Überabschätzung der tatsächlichen Fehler kommt durch die Vernachlässigung der Dateninformationen. Aus diesem Grund vollzieht *mojo* nach der Analyse von Deadlocks und Synchronisierungsfehlern eine *Prädikatenanalyse*. Dabei leitet der Compiler Zusicherungen ab. Solch eine Zusicherung $x = assert(y, predicate)$ steht für eine einfache Kopieroperation ($x = y$), garantiert jedoch zusätzlich, dass das Prädikat *predicate* für den Wert von y wahr ist. Dadurch werden (ähnlich zu ϕ- und π-Funktionen) zusätzliche Kopien einer Variable erzeugt.

Die Prädikate erster Ordnung ergeben sich aus der Menge der Zuweisungen und Bedingungen. Gelten mehrere Prädikate für eine einzelne Instruktion, so werden diese konjunktiv zusammengeschlossen. Gibt es mehrere Definitionen auf exklusiven Pfaden für dieselbe Variable, so werden diese Prädikate disjunktiv verknüpft: Das Prädikat einer Variablen liegt in disjunktiver Normalform vor.

Der Vorteil der Zwischenrepräsentation und der Verwendung der CSSA-Form liegt in der einmaligen Definition von Variablen. Aus diesem Grund kann der Zustandsraum einer Variablen einmalig direkt bei seiner Definition annotiert werden. Durch die zusätzliche Einführung von Zusicherungen wird dies (ähnlich zur Static Single Information (SSI) Form [21]) auch nach Verzweigungen, etc. möglich, wo sich der tatsächliche Wert der Variablen nicht ändert, sondern nur dessen garantierter Zustandsraum. Zur Ableitung der Prädikate mittels Datenflussanalyse verweisen wir an dieser Stelle auf unsere Vorarbeiten [22].

Mit Hilfe der Prädikate überprüft *mojo*, ob bspw. ein als nicht aktivierend eingestufter Knoten eines AND-Joins unter den Zusicherungen doch ein Aktivierungsknoten ist. Diese Überprüfung findet unter Zuhilfenahme des SMT-Lösers *SMTInterpol* [23] statt.

Insgesamt kann die Abschätzung der Fehler mit Hilfe der Prädikatenanalyse präzisiert werden. Wir geben die eliminierten Fehler dennoch als Warnung aus, da sie nur durch den bedingten Kontrollfluss verhindert werden.

Nicht lebendige Knoten Die Prädikatenanalyse wird auch zur Warnung des Modellierers über *nicht lebendige Knoten* genutzt. Ein nicht lebendiger Knoten ist ein Knoten des Prozesses, der aufgrund der Bedingungen, die auf jedem Pfad zu ihm vorhanden sind, *niemals* ausgeführt werden kann. Dies ist vergleichbar mit *Deadcode Elimination*.

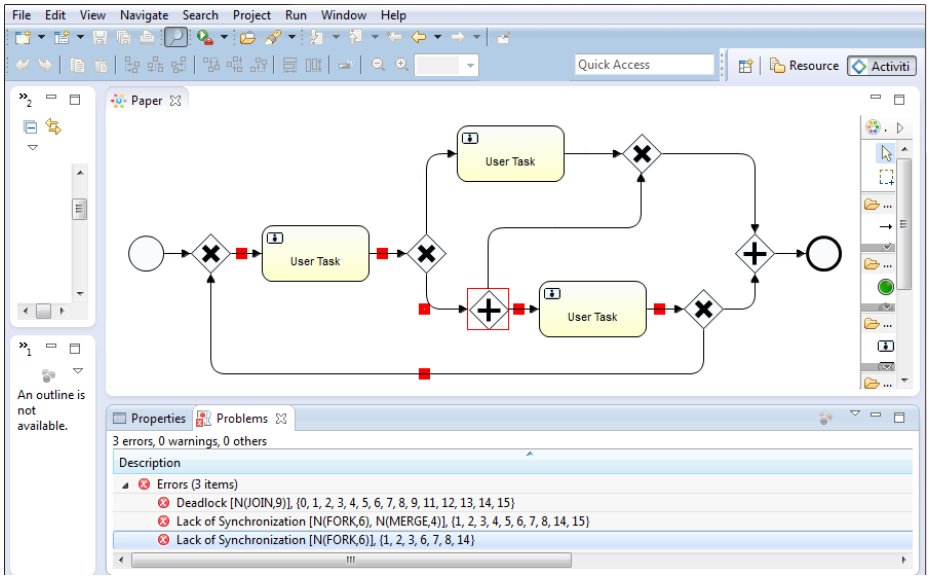

Abbildung 8. Fehlerdiagnose im Activiti Designer [18]

2.5 Backend

Das Backend von *mojo* erfüllt zwei Aufgaben: Die Visualisierung von Fehlern im Prozessmodellierungswerkzeug und die Linearisierung des Foldoutgraphen bei festgestellter Korrektheit des Prozesses.

Das Backend ist abhängig von der Workflowengine, die den Prozess ausführen soll, als auch vom gewählten Modellierungstool. Zum Zeitpunkt dieser Arbeit wurde *mojo* in den *Activiti Designer* [2] integriert. Der Designer nutzt sowohl den Compiler in jedem Motivierungsschritt des Prozesses als auch die von dem Compiler zurückgegebenen Fehlerinformationen. Diese Fehlerinformationen werden dann in einem speziellen Fenster angezeigt und direkt im Prozess illustriert. In einem Fehlerdiagnosemodus können die Fehler im Einzelnen untersucht werden. Dabei werden viele Informationen über den Fehler, wie der verursachende Knoten, die Fehlerwirkung und eine Fehlerbeschreibung graphisch im Prozess hervorgehoben. Abbildung 8 zeigt eine Bildschirmaufnahme der Applikation. Eine solch detailierte Fehlerdiagnose und die vielen verschiedenen Arten von Fehlern sind in der Prozessmodellierung einzigartig.

Der Kodierer vollführt derzeit eine einfache Linearisierung des Foldoutgraphen aus Mangel einer entsprechend allgemeingültigen virtuellen Maschine (Prozessengine). An dieser Stelle sind Techniken, wie sie in dem SafeTSA-Format [14] verwendet werden, angedacht: Ein inhärent typ- und referenzsicheres und damit einfach zu verfizierendes mobiles Format. Damit wird sowohl die Verifikation und das Einlesen von Prozessen als auch die Ausführung beschleunigt.

3 Ausblick

In dieser Arbeit haben wir unseren Compiler *mojo* vorgestellt. Er ist der erste Compiler für (technische) Geschäftsprozesse. Er bietet zahlreiche Features, die in der Analyse und Transformation von Prozessen heute einzigartig ist: *1)* Vollständige Unterstützung von OR-Forks und OR-Joins, *2)* Schnell analysierbare Zwischenrepräsentation, *3)* Anzeigen von Wettkampfsbedingungen, *4)* Analyse von Gültigkeitsbereichen von Variablen, *5)* CSSA-Form, *6)* Vollständige Auflistung aller Verklemmungen und Synchronisierungsfehler, *7)* Einbeziehung von Dateninformationen, *8)* Analyse von toten Knoten und *9)* Detailierteste Fehlerdiagnostik mit Fehlerursache und Fehlerwirkung.

In zukünftigen Arbeiten soll *mojo* zur Produktreife für die Forschung weiterentwickelt werden. Dabei gehören auch weitere Analysen, wie das Auffinden von Zugriffen auf vorher gelöschte Variablen, Adaption der Techniken zur Kontrollflussentfaltung aus unseren früheren Arbeiten [22] und Korrekturvorschläge. Zudem soll die Veröffentlichung der neuen Techniken durch einen quelloffenen Download stattfinden.

Literatur

1. OMG: Business Process Model and Notation 2.0. formal/2011-01-03 (2011)
2. Alfresco: Activiti BPM Platform. `http://activiti.org/` (last access: February 18, 2015)
3. redhat: jBPM - Open Source Business Process Management - Process engine. `http://www.jbpm.org/` (last access: February 18, 2015)
4. IBM: IBM WebSphere software - United States. `http://www.ibm.com/software/websphere/` (last access: February 18, 2015)
5. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to petri nets. In van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings. Volume 3649. (2005) 220–235
6. Verbeek, H.M.W.E., van der Aalst, W.M.P.: Woflan 2.0: A petri-net-based workflow diagnosis tool. In: ICATPN. (2000) 475–484
7. Prinz, T.M., Heinze, T.S., Amme, W., Kretzschmar, J., Beckstein, C.: Towards a compiler for business processes - a research agenda. In de Barros, M., Rückemann, C.P., eds.: SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing. Volume 6., Nice, France, IARIA Conference, ThinkMind Digital Library (March 22 2015) 49–54
8. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The petri net markup language: Concepts, technology, and tools. In: ICATPN. (2003) 483–505
9. Amme, W., Martens, A., Moser, S.: Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. International Journal of Business Process Integration and Management **4**(1) (2009) 47–59
10. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent static single assignment form and constant propagation for explicitly parallel programs. In Li, Z., Yew, P., Chatterjee, S., Huang, C., Sadayappan, P., Sehr, D.C., eds.: Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC'97, Minneapolis, Minnesota,

USA, August 7-9, 1997, Proceedings. Volume 1366 of Lecture Notes in Computer Science., Springer (1997) 114–130
11. Spiess, N.: Realisation of a framework for the analysis of business processes (2013)
12. Prinz, T.M., Amme, W.: A complete and the most liberal semantics for converging or gateways in sound processes. Complex Systems Informatics and Modeling Quarterly, CSIMQ **Issue 4** (Oct 2015) [To be published].
13. Crelier, R.: OP2: A Portable Oberon Compiler. Technical Report ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Computer Systeme 125, Eidgenössische Technische Hochschule Zürich, Zurich, Switzerland (feb 1990)
14. Amme, W., von Ronne, J., Franz, M.: Ssa-based mobile code: Implementation and empirical evaluation. TACO **4**(2) (2007)
15. Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time. In Sarkar, V., Ryder, B.G., Soffa, M.L., eds.: Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994, ACM (1994) 171–185
16. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In Dumas, M., Reichert, M., Shan, M., eds.: Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings. Volume 5240 of Lecture Notes in Computer Science., Springer (2008) 100–115
17. Prinz, T.M.: Proposals for a virtual machine for business processes. In Heinze, T.S., Prinz, T.M., eds.: Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19-20, 2015. Volume 1360 of CEUR Workshop Proceedings., CEUR-WS.org (2015) 10–17
18. Prinz, T.M., Amme, W.: Practical compiler-based user support during the development of business processes. In Lomuscio, A., Nepal, S., Patrizi, F., Benatallah, B., Brandic, I., eds.: Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers. Volume 8377 of Lecture Notes in Computer Science., Springer (2013) 40–53
19. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. **25**(2) (2000) 117–134
20. Prinz, T.M., Spieß, N., Amme, W.: A first step towards a compiler for business processes. In Cohen, A., ed.: Compiler Construction - 23rd International Conference, CC 2014, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Volume 8409 of Lecture Notes in Computer Science., Springer (2014) 238–243
21. Ananian, C.S.: The static single information form. Master's thesis, Massachusetts Institute of Technology (MIT), Massachusetts (Sep 1999)
22. Heinze, T.S., Amme, W., Moser, S.: Compiling more precise petri net models for an improved verification of service implementations. In: 7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014, IEEE Computer Society (2014) 25–32
23. Christ, J., Hoenicke, J., Nutz, A.: Smtinterpol: An interpolating SMT solver. In Donaldson, A.F., Parker, D., eds.: Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. Volume 7385 of Lecture Notes in Computer Science., Springer (2012) 248–254