RWTH Aachen

Department of Computer Science
Technical Report

Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

René Wörzberger

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2010-04

RWTH Aachen · Department of Computer Science · April 2010

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines Doktors der
Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
René Wörzberger

aus Hannover

Berichter: Universitätsprofessor Dr.-Ing. Manfred Nagl
Universitätsprofessor Dr. rer. nat. Gregor Engels
Universitätsprofessor Dr. rer. nat. Bernhard Rumpe

Tag der mündlichen Prüfung: 11. Januar 2010

Die Druckfassung dieser Dissertation
ist unter der ISBN 978-3-8322-8949-2 erschienen.

Kurzfassung

Prozessmanagementsysteme dienen der Unterstützung von Geschäftsprozessen. Sie versorgen dabei Prozessbeteiligte, wie beispielsweise Sachbearbeiter oder Prüfsysteme in einer Versicherung, zur richtigen Zeit mit relevanten Daten.

Komplexe Geschäftsprozesse sind nicht vollständig vor ihrer Durchführung planbar. *Unvorhersehbare Ereignisse* erfordern, dass von der in einem Prozessmodell festgehaltenen Planung während der Prozessdurchführung *abgewichen* wird. Diese Eigenschaft von Geschäfts- und Entwicklungsprozessen wird als *Dynamik* bezeichnet.

Gängige Prozessmanagementsysteme unterstützen die Dynamik in Geschäftsprozessen nur *unzureichend*. Sie setzen voraus, dass ein Geschäftsprozess vorab vollständig modelliert wird und legen somit vor der Durchführung mögliche Abläufe fest. *Abweichungen* vom Prozessmodell sind in ihnen *nicht vorgesehen*. Folglich ergeben sich hochkomplexe, schlecht wartbare Prozessmodelle, die jedoch dem Anspruch ebenfalls nicht genügen, jeden sinnvollen Prozessablauf abzudecken. Ergibt sich aus fachlichen Notwendigkeiten bei der Prozessdurchführung ein Ablauf, den das jeweilige Prozessmodell nicht vorsieht, müssen die Prozessbeteiligten den Prozess entkoppelt von Prozessmanagementsystem weiterführen. Das Prozessmanagementsystem kann dann jedoch nicht mehr zur Unterstützung des Prozesses beitragen.

In *dieser Arbeit* wird ein Prozessmanagementsystem beschrieben, das auch dynamische Prozesse unterstützt. Dynamische Änderungen können von Prozessbeteiligten über strukturelle *Änderungen* an Modellen bereits *laufender Prozesse* durchgeführt werden.

Im Gegensatz zu verwandten Arbeiten wurde das Prozessmanagementsystem nicht von Grund auf neuentwickelt. Stattdessen *erweitert* es das kommerzielle System IBM WebSphere Process Server, das von der Generali Deutschland Informatik Services GmbH eingesetzt wird – dem kooperierenden Industriepartner dieser Arbeit. Hierdurch kann die *vorhandene Funktionalität* des Process Servers *weiterverwendet* werden und muss nicht reimplementiert werden.

Es wurden folgende *Ergebnisse* im Einzelnen erzielt:

Die unvollständige *Modellwelt* des Process Servers wurde *vervollständigt*: *Prozessdefinitionsmodelle* in der Standardsprache WS-BPEL werden ergänzt um speziellere *Prozessinstanzmodelle*, die zusätzliche Informationen über den aktuellen *Ausführungszustand* eines konkreten Prozesses und dessen bisherige *Prozesshistorie* beinhalten. Abstrakter als Prozessdefinitionsmodelle sind so genannte *Prozesswissensmodelle*, in denen *grobe Festlegungen* über Abläufe in Prozessen getroffen werden. Dazu gehören Festlegungen bzgl.

Reihenfolgen bestimmter Aktivitäten, deren gegenseitiger *Ausschluss* sowie Ausführungshäufigkeiten bestimmter Aktivitäten in bestimmten Prozessen. Sowohl die *Syntax* als auch die *Semantik* dieser drei Prozessmodellarten wurde *formal* mithilfe von Metamodellen bzw. Graphersetzungssystemen, Transitionssystemen und temporalen Logikformeln beschrieben. Ein modellgetrieben entwickelter *Prozessmodelleditor* dient zur Darstellung und Editierung von Modellen aller drei Arten.

Die Semantikdefinition der Prozessdefinitions- und -instanzmodelle ist Ausgangspunkt für die Realisierung einer den WebSphere Process Server erweiternden, so genannten *Dynamik-Schicht*. Diese *simuliert dynamische Änderungen*, die direkt im Process Server nicht vorgenommen werden können. Die Simulation ist opak für Prozessbeteiligte, die dynamische Änderungen als strukturelle Änderungen in Prozessinstanzmodellen innerhalb des Prozessmodelleditors erfahren.

Dynamische Änderungen unterliegen *technischen und fachlichen Rahmenbedingungen*. Ein *syntaxbasiertes* und im Prozessmodelleditor implementiertes *Prüfwerkzeug* stellt sicher, dass Modelländerungen und insbesondere dynamische Änderungen an Prozessinstanzmodellen, diese Rahmenbedingungen nicht verletzen. Im Vergleich zu verwandten Arbeiten, die technische und fachliche Regeln getrennt behandeln, wurde in dieser Arbeit ein *einheitlicher Ansatz* gewählt, der beide Regelarten berücksichtigt und auf der Object Constraint Language (OCL) basiert. Das heißt, dass alle Modellarten auf innere (technische) Korrektheit geprüft werden können, aber auch Prozessinstanzmodelle auf Komplianz zu *expliziten*, fachlichen Rahmenbedingungen überprüft werden können, die in Prozesswissensmodellen festgehalten sind.

Prüfungen gegen in Prozesswissensmodellen explizit modelliertes Prozesswissen wurden durch Prüfungen gegen *implizites Prozesswissen* ergänzt. Unter implizitem Prozesswissen wird in diesem Zusammenhang Wissen über Ausführungshäufigkeiten und -reihenfolgen von Aktivitäten verstanden, das sich in Prozessdefinitions- und -instanzmodellen indirekt wiederfindet. In unterschiedlichen Prozessinstanz- und -definitionsmodellen inkonsistent modellierte Abläufe werden durch ein weiteres *Prüfwerkzeug* aufgedeckt. Dieses Prüfwerkzeug basiert auf einer *Graphgrammatik*, die zwei Prozessinstanzmodelle – soweit möglich – simultan ausführt und dabei erreichbare Ausführungszustände in einem Transitionssystem festhält. Dieses Transitionssystem ist Ausgangspunkt für einen *Modellvergleich*, der Prozessbeteiligten *detaillierte Informationen* über inkonsistente Abläufe in unterschiedlichen Prozessinstanzmodellen liefert.

Danksagungen

Viele liebe Menschen haben mich während meiner Promotion unterstützt und zur Entstehung dieser Dissertation beigetragen. Hierfür möchte ich mich bedanken.

Prof. Dr.-Ing. M. Nagl verdanke ich die wissenschaftliche Betreuung und natürlich auch die finanzielle Ausstattung. Von ihm habe ich in den vergangenen Jahren viel gelernt. Ich wünsche Ihm, dass er – nach den zahlreichen zusätzlichen Aufgaben zum Wohle der deutschen Universität – bald die Zeit findet, sich auch anderen Herzensangelegenheiten zu widmen. Prof. Dr. G. Engels danke ich für die Übernahme des Koreferats meiner Dissertation. Durch ihn "grooved" die Arbeit sicher mehr, als sie es ohne ihn getan hätte. Der neue Lehrstuhlinhaber Prof. Dr. B. Rumpe hat für viele neue Ideen (und natürlich Kollegen) gesorgt. Er war zur Stelle, als durch die offizielle Exegese der Promotionsordnung ein dritter Gutachter notwendig wurde. Prof. Dr. J. Giesl, Prof. Dr. W. Thomas und Prof. Dr. L. Kobbelt danke ich, dass Sie als weitere Prüfer zur Verfügung standen. Ebenso gebührt Dank meinen Kooperationspartnern K. Wolf, H. Wessels und Dr. S. Bühne von der Generali Deutschland Informatik Services GmbH. Der Deutschen Forschungsgemeinschaft danke für die finanziellen Mittel, die ich in meinem Projekt in Anspruch nehmen konnte.

Ich habe in den vergangenen Jahren mit vielen netten Kollegen arbeiten dürfen. Oliver und Felix haben mir die Tür zum Lehrstuhl für Informatik 3 geöffnet. Möge Sie der Herrgott/Spielleiter dafür mit Gesundheit/Hitpoints segnen. Simon hat meine Diplomarbeit betreut und war mir anschließend ein wunderbarer Büronachbar. Ihm verdanke ich Erinnerungen an unsere "Dialektwochen", Unterhaltung bei gemeinsamen Klausurkorrekturmarathons und die abendliche, kalorienneutrale Sportausübung. Markus und Thomas Haase möchte ich für die tolle Zusammenarbeit in der anstrengenden SFB-Begehungsphase danken. Michael und Christof danke ich für die Einblicke in das Leben nach der Promotion, Marita, Galina, Boris, André M., Ulrich, Manfred, Dirk, Ansgar und Bernhard W. für die schönen frühen Jahre meines Lehrstuhlaufenthalts. Bodo danke ich, dass er mir die Softwaretechnik-Vorlesung für MaTAs inklusive vollständiger Unterlagen vererbt hat. Angelika und Silke schulde ich Dank für die Hilfe bei vielen organisatorischen Aufgaben. Ulrike und Erhard danke ich für das tolle Teamwork bei diversen Lehrveranstaltungen und amüsante Konferenzaufenthalte ("Now you may talk!"). Theresa verdanke ich, dass sie in mir ~~den Spaß~~ ein gewisses Interesse an Dauerläufen geweckt hat. Ich bin überzeugt, dass Sie Ihren Kollegen auch in Zukunft ein hilfreicher Tempomacher sein kann. Bei Thomas Heer möchte ich mich für die tolle Zusammenarbeit bedanken, die unseren wissenschaftlichen Output deutlich erhöht hat. Viele semantisch tiefe und konstruktive Anmerkungen

zu meiner Dissertation stammen von ihm. Ibrahim schulde ich die Einsicht, dass ich tatsächlich kein bisschen Fußball spielen kann. Durch sein “Baby” – den Info-Cup – konnte ich das eindrucksvoll vor der Fachgruppe demonstrieren. Dank Daniel, der seine Promotion gleichzeitig zu meiner erfolgreich abgeschlossen hat, konnte ich mein Leid in der Endphase teilen. Unsere Umläufe haben sich ein spannendes “Rennen” geliefert. Cem danke ich für seine Hilfsbereitschaft. Seinen stahlharten Nerven konnten im Gegensatz zu meinen auch Konferenzvorbereitungen in luftiger Höhe nichts anhaben. Den neuen Kollegen aus Braunschweig Jan, Claas, Christoph, Martin, Arne und Christian danke ich für den frischen Wind, den sie mitgebracht haben. Sie haben der alten Garde und insbesondere mir in der Schlussphase viel Arbeit abgenommen.

Ich durfte während meiner Promotion einige liebe Diplomanden betreuen. Rui und Salman haben mich durch ihre Leistungen unter zeitlichem Druck beeindruckt, Frank und Alexej durch ihre Selbständigkeit. Nicolas verdanke ich die Zuarbeit bei der Erweiterung des Prozessmanagementsystems und viele interessante Gespräche über Politik, Philosophie und Respawn-Spots. Thomas K. hat immer brav mit mir Kaffee getrunken und sich für mich mit unfertigen Eclipse-Rahmenwerken herumgeärgert. Für Thomas L. gilt das Gleiche. Über ihn habe ich viel darüber erfahren, wie man ein Doppelstudium mit HiWi-Tätigkeit und Familie in Einklang bringt. André F. war viele Jahre ein ausgesprochen pflichtbewusster, engagierter und zuverlässiger HiWi und Diplomand. Ohne seine Mitarbeit hätten meine Prototypen sicher nicht ihren heutigen Stand erreicht.

Auch außerhalb der RWTH Aachen gibt es Leute, denen ich Dank schulde. Meine Freunde haben für das manchmal nötige Maß an Zerstreuung gesorgt und mir eine gewisse Präsenzlosigkeit in letzter Zeit nachgesehen. Siggi, Karola, Adi und Helga haben mich die letzten Jahre mit Pfeffersteaks, Croissants und (manchmal) Kölsch versorgt. Hierfür und für viele andere Dinge danke ich ihnen. Meine Familie hat mich in der manchmal anstrengenden Zeit sehr unterstützt. Ohne sie hätte ich die Promotion vermutlich nicht so erfolgreich abschließen können. Hierfür möchte ich mich bedanken. Mein ganz besonderer Dank gebührt Alien. Sie hat mir meine gelegentliche Entrücktheit stets verziehen. Ohne ihre Unterstützung und Liebe wäre die Zeit nicht halb so schön gewesen. Ich freue mich auf noch viele wunderschöne Jahre mit ihr.

Aachen, 19. Januar 2010 *René Wörzberger*

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Der Gewinn eines Unternehmens wird nur zum Teil durch die Qualität der angebotenen Produkte bzw. Dienstleistungen bestimmt. Ebenso wichtig ist es, die Prozesse, über die die Produkte hergestellt bzw. die Dienstleistungen erbracht werden, einerseits kostengünstig und andererseits offen für Veränderungen zu gestalten. Um dies zu erreichen, bedarf es eines geeigneten Prozessmanagements.

Es gibt eine Vielzahl von Mitteln, die dem Management von Prozessen dienen. Dazu gehören Sprachen, mittels derer Prozesse modelliert werden können. Die resultierenden Prozessmodelle sind die Grundlage für eine systematische Prozessverbesserung. Abhängig von der verwendeten Sprache können die Prozessmodelle auch zur Steuerung eines konkreten Prozesses durch ein Prozessmanagementsystem gebraucht werden. Dieses nutzt die in einem Prozessmodell vorhandenen Vorgaben, um Buch über den Prozessfortschritt zu führen, Entscheidungen über den weiteren Prozessverlauf zu treffen und Prozessbeteiligte zum richtigen Zeitpunkt mit den richtigen Informationen zu versorgen.

Die vorliegende Arbeit befasst sich mit der informatischen Unterstützung des Managements von Geschäftsprozessen durch Prozessmanagementsysteme. Das Hauptaugenmerk der Arbeit liegt auf der angemessenen Behandlung der Dynamik, die vielen Geschäftsprozessen innewohnt. Hierbei ist dem Umstand Rechnung zu tragen, dass sich Geschäftsprozesse mitunter im Vorfeld nicht vollständig planen lassen bzw. von einer Vorplanung während der Durchführung abweichen.

Eine wesentliche Randbedingung der in dieser Arbeit geschaffenen Konzepte ist die Berücksichtigung industrieller Standards und Systeme. Diese Bedingung wurde durch den Projektpartner des Forschungsprojekts, Generali Deutschland Informatik Services GmbH (GDIS), formuliert. Die GDIS ist der IT-Dienstleister der Generali Gruppe – ein Allsparten-Versicherungskonzern. Die entwickelten Konzepte und Prototypen setzen daher auf dem von der GDIS eingesetzten Prozessmanagementsystem WebSphere Process Server [WAM^{+}07] auf. Dieses verarbeitet wiederum Prozessmodelle in der Standard-Modellierungssprache WS-BPEL (Web Services Business Process Execution

Language) [AAA+07], welche daher die Basis für die in dieser Arbeit geschaffenen, ergänzenden Prozessmodellierungssprachen bildet.

Dieses Kapitel widmet sich in Abschnitt 1.1 zunächst der Klärung grundlegender Fachbegriffe und der Abgrenzung betrachteter Geschäftsprozesse von anderen Prozessarten. In Abschnitt 1.2 wird die derzeit gängige, unzureichende Unterstützung der Dynamik in Prozessmanagementsysteme beschrieben. In Abschnitt 1.3 wird der in dieser Arbeit verfolgte Lösungsansatz an einem durchgängigen Beispiel konkretisiert. Abschnitt 1.4 identifiziert aus dem Ansatz folgende Fragestellungen, die in dieser Arbeit beantwortet wurden und fasst die wissenschaftlichen Beiträge dieser Arbeit kurz zusammen.

1.1 Grundlagen und Begriffe

Im Folgenden werden zunächst grundlegende Begriffe und Aspekte im Bereich des Prozessmanagements erläutert.

1.1.1 Prozessbegriff

Der Begriff *Prozess* ist vielfach belegt. Gemein ist allen Bedeutungen, dass sie zumindest mittelbar einen Verlauf [Mac77] bzw. synonym einen Vorgang, Hergang, Fortgang, Entwicklung oder Ablauf bezeichnen [RS82]. Die etymologischen Wurzeln liegen im lateinischen *processus* (Fortgang), welches wiederum von *procedere* (vorwärts gehen) ableitet [KS02].

Ein *Prozess* bezweckt die Erstellung bzw. Veränderung eines materiellen Produkts oder Informationsbestands. Ein Prozess überführt einen Systemzustand dabei schrittweise in einen anderen. Beispielsweise überführt der Produktionsprozess eines PKWs Einzelteile in den zusammengesetzten und einsatzfähigen PKW. Ein Softwareentwicklungsprozess setzt (häufig vage) Anforderungen in einsetzbare Software um. Geschäftsprozesse, beispielsweise die Schadenabwicklung im Versicherungsumfeld, führen zur Anreicherung bzw. Veränderung von Informationen, z. B. Informationen über den Schadenfreiheitsrabatt eines Kunden oder Buchhaltungsdaten des Versicherungsunternehmens durch Auszahlung von Entschädigungssummen [All05, Abschnitt 2.2].

1.1.2 Prozessaspekte

Prozesse können unter verschiedenen Aspekten betrachtet werden. Wichtige Aspekte sind u. a. [JB96, Kapitel 6] und [NW94, Kapitel 4]

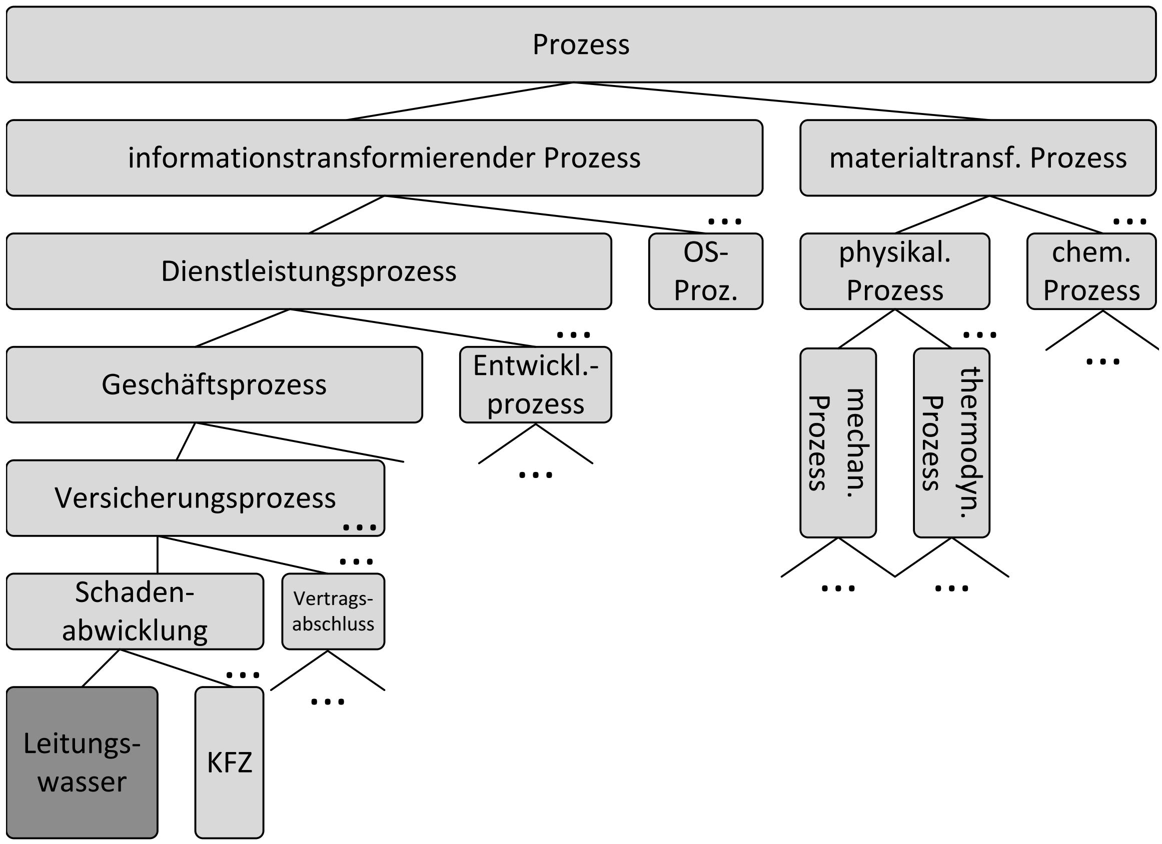


Abbildung 1.1: Klassifizierung verschiedenartiger Prozesse

- der *Aktivitätsaspekt*, der die mitunter mehrstufige Dekomposition eines Prozesses in einzelne Aktivitäten und Reihenfolgebeziehungen sowie Ausführungsbedingungen der Aktivitäten und die Kontrollflussbeziehungen der Aktivitäten untereinander fokussiert,

- der *Produktaspekt*, der die in einem Prozess entstehenden und veränderten materiellen oder immateriellen Produkte in verschiedenen Versionen betrachtet und

- der *Organisationsaspekt*, in dessen Mittelpunkt die in einem Prozess eingesetzten menschlichen oder technischen Ressourcen stehen.

Hinzu kommen Querbezüge zwischen diesen Aspekten, beispielsweise durch die Zuordnung eines Mitarbeiters (menschliche Ressource) zu einer Aktivität.

1.1.3 Prozessarten

In dieser Arbeit wird nur die spezielle Klasse der Geschäftsprozesse behandelt und dabei insbesondere Versicherungsprozesse. Die folgenden Abschnitte klassifizieren verschiedenartige Prozesse und dienen somit der Charakterisierung der Geschäftsprozesse im Vergleich zu anderen Prozessen. Die Erläuterungen orientieren sich an Abbildung 1.1. Diese zeigt eine Taxonomie für Prozesse. Die Hierarchie ist allerdings nur eine von vielen möglichen und nicht vollständig.

Geschäftsprozesse sind demnach generell informationstransformierend, wandeln also abstrakte Systemzustände (z.B. Versicherungsvertragsverhältnisse) ineinander um oder erweitern vorhandene Informationen, überführen aber beispielsweise keine Materialien in andere wie chemische Prozesse.

Versicherungsprozesse lassen sich selbst weiter unterklassifizieren. Typische Versicherungsprozesse sind beispielsweise Prozesse, über die ein Versicherungsvertrag zustande kommt. Getrennt davon werden in Versicherungen Prozesse durchgeführt, deren Zweck die Abwicklung eines Schadensfalls (Schadenregulierung) ist. Hierbei wird typologisch weiter nach Schadenart unterschieden. Die Regulierung eines KFZ-Schadens verläuft generell etwas anders als die eines Leitungswasserschadens. Letztgenannter Prozesstyp wird in Abschnitt 1.3 näher erläutert und dient als durchgängiges Beispiel dieser Arbeit.

1.1.4 Prozesse und Prozessmodelle

Die Abbildung 1.2 fasst wesentliche Begriffe und deren Zusammenhänge aus dem Prozessmanagement zusammen. Im Folgenden werden diese einzeln erläutert.

Im Folgenden wird zwischen Prozessen, Prozesstypen und Prozesswissen unterschieden. Ein Prozess wird als einmalig und unterscheidbar von anderen Prozessen angesehen, wohingegen ein *Prozesstyp* eine Klasse von gleichartigen Prozessen repräsentiert. Ein Beispiel für einen Prozess wäre die “Regulierung des KFZ-Haftpflichtschadens mit Schadennummer 1234”. Der zugehörige Prozesstyp würde dementsprechend “KFZ-Schadenregulierung” lauten. Prozesswissen abstrahiert von konkreten Typen und umfasst domänenspezifisches Wissen über Prozesse, beispielsweise, welche Aktivitäten generell einander bedingen oder ausschließen.

Ein *Prozessmanagementsystem* verwaltet *Prozessmodelle*, die Abstraktionen von Prozessen, Prozesstypen bzw. allgemeinem Prozesswissen darstellen. Eine Unterscheidung zwischen dem abstrakten, gedanklichen Modell und dessen Darstellung wie in [JBS99] wird in dieser Arbeit nicht vorgenommen. Die Unterstützung, die ein Prozessmanagementsystem bereitstellt, umfasst im Wesentlichen die Modellierung von Prozesstypen als Prozessdefinitionsmodelle und die Steuerung eines konkreten Prozesses mittels eines Prozessinstanzmodells.

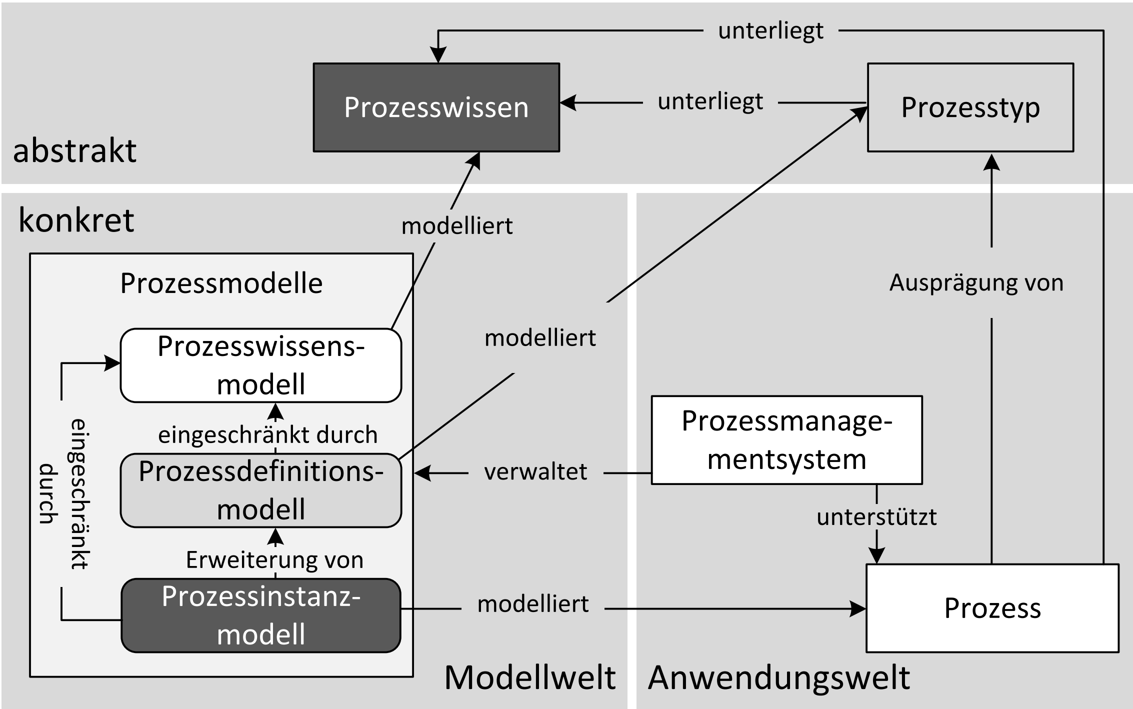


Abbildung 1.2: Begriffe des Prozessmanagements (nach [JBS99])

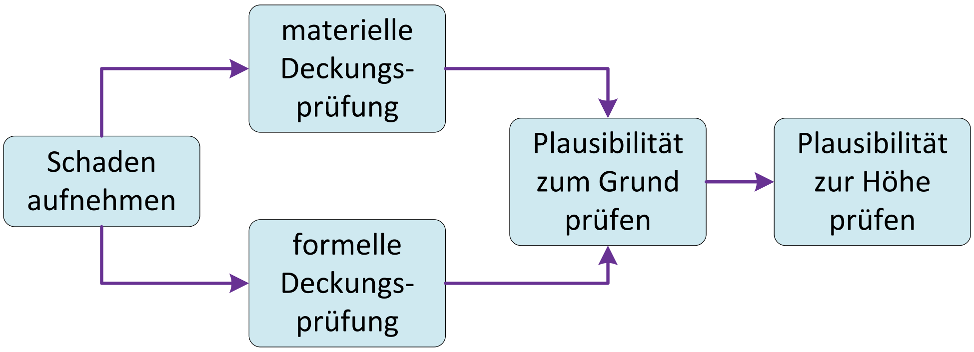


Abbildung 1.3: Einfaches Beispiel eines Prozessdefinitionsmodells

Prozessdefinitionsmodell

In einem *Prozessdefinitionsmodell* wird ein Prozesstyp modelliert. Die Syntax und Semantik von Prozessdefinitionsmodellen wird durch eine Prozessmodellierungssprache festgelegt, von denen eine Vielzahl existiert [DAH05, Hav05]. In den meisten Prozessmodellierungssprachen werden die Modelle diagrammatisch dargestellt. Einige Prozessmodellierungssprachen dienen nur zur Modellierung bestimmter Aspekte eines Prozesses. Dazu gehört im Allgemeinen immer der Aktivitätsaspekt, d.h. aus einem Prozessdefinitionsmodell geht hervor, welche Aktivitäten in einem Prozess (ggf. optional oder iterativ) in welcher Reihenfolge ausgeführt werden.

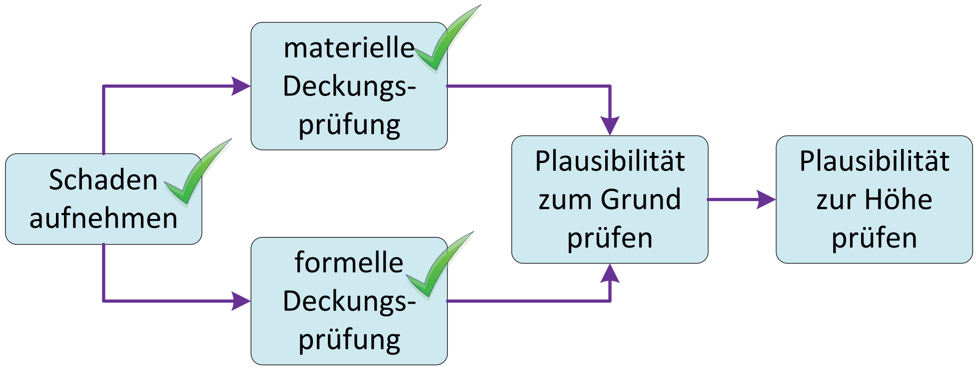


Abbildung 1.4: Einfaches Beispiel eines Prozessinstanzmodells

Beispiel 1.1 (Prozessdefinitionsmodell) Abbildung 1.3 zeigt ein Beispiel eines Prozessdefinitionsmodells, das vereinfacht einen Ausschnitt einer typischen Schadenregulierung modelliert. Die Prozessmodellierungssprache ist eine vereinfachte Variante der später in dieser Arbeit verwendeten Sprache für Prozessdefinitionsmodelle (vgl. Abschnitt 2.2). In diesem Beispiel sollen alle Aktivitäten durch menschliche Sachbearbeiter durchgeführt werden. Die Verbindungen zwischen den Aktivitäten sind Kontrollflussdefinitionen, über die mögliche Reihenfolgen der Aktivitäten festgelegt werden. In diesem Fall beginnt der Prozess also mit der Schadenaufnahme gefolgt von der materiellen und formellen Deckungsprüfung, die in beliebiger Reihenfolge ausgeführt werden können. Nach Abschluss der Deckungsprüfungen wird die Plausibilität der Schadensmeldung überprüft und zwar zuerst dem Grunde nach und anschließend zur Höhe.

Prozessinstanzmodell

So wie ein Prozesstyp eine *Abstraktion* eines konkreten Prozesses darstellt und umgekehrt ein Prozess eine konkrete *Ausprägung* eines bestimmten Prozesstyps darstellen kann, ist ein *Prozessinstanzmodell* von einem Prozessdefinitionsmodell abgeleitet. Ein Prozessinstanzmodell beinhaltet zusätzlich zum Prozessdefinitionsmodell instanzspezifische Informationen. Dazu gehören Zustandsinformationen einzelner Aktivitäten, Ressourcenzuweisungen und (Verweise auf) produzierte Daten, die dem Produktaspekt zuzurechnen sind. Ein Prozessinstanzmodell ist zu jedem Zeitpunkt eines bestimmten Prozesses ein Abbild des Zustands, in dem sich der Prozess befindet.

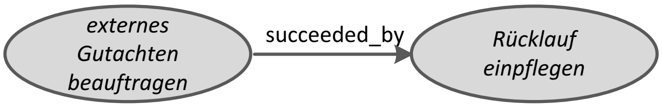


Abbildung 1.5: Einfaches Beispiel eines Prozesswissensmodells

Beispiel 1.2 (Prozessinstanzmodell) Die Abbildung 1.4 stellt ein Prozessinstanzmodell dar. Sie ist vom Prozessdefinitionsmodell aus Abbildung 1.3 abgeleitet. Der Einfachheit halber sind im Prozessinstanzmodell Ressourcen und Daten nicht dargestellt, sondern nur Zustände von Aktivitäten. Der Haken symbolisiert, dass eine Aktivität bereits ausgeführt und abgeschlossen wurde, in diesem Fall also die Schadenaufnahme und die Deckungsprüfungen. Die Plausibilitätsprüfung zum Grund ist in diesem Beispiel gerade in Bearbeitung. Die Prüfung der Plausibilität zur Höhe ist noch nicht gestartet.

Prozesswissensmodelle

Prozesswissensmodelle modellieren allgemeines Prozesswissen, z.B. generelle Rahmenbedingungen für Prozesse in einer bestimmten Anwendungsdomäne. Dieses Wissen ist abstrakter als das Wissen, das bereits in Prozessdefinitionsmodellen über Prozesstypen vorhanden ist.

Beispiel 1.3 (Prozesswissensmodell) In Abbildung 1.5 ist ein sehr einfaches Prozesswissensmodell dargestellt. Es ist in einer vereinfachten Variante der in dieser Arbeit entwickelten Sprache für Prozesswissensmodelle notiert (vgl. Unterabschnitt 2.2.3). Das Modell besagt, dass der Rücklauf eines beauftragten Gutachtens eingepflegt werden muss und zwar nachdem das Gutachten beauftragt wurde.

1.1.5 Dynamik in Prozessen

Die in dieser Arbeit zentrale Prozesseigenschaft ist die *Dynamik* von Prozessen. Dynamik wird dabei im engeren Sinne aufgefasst. Sie bezeichnet nicht die in jedem Prozess trivialerweise vorhandene Dynamik, die daher rührt, dass ein Prozess immer einen Veränderungsvorgang bezeichnet. Stattdessen bezieht sich Dynamik hier auf die Planbarkeit von Prozessen. Ein dynamischer Prozess ist nur schlecht planbar. Antonym zur Dynamik wird in Bezug auf Prozesse der Begriff *Statik* verwendet.

Die *Planbarkeit* von Prozessen wird dadurch bestimmt, inwieweit die für den Prozess relevanten *Eingaben* bekannt sind. Zu den Eingaben gehören Vorgaben, die besagen

➢ welche *Ergebnisse*, also welche Systemveränderungen oder Ausgabeinformationen erzielt werden müssen (z.B. medizinische Gutachten in einem Versicherungsprozess oder die Entwicklerdokumentation in einem Softwareentwicklungsprozess). Die Vorgaben hierzu können von Prozessbeteiligten stammen (z.B. Geschädigter oder Versicherter in einem Versicherungsprozess oder Auftraggeber in einem Entwicklungsprozess) oder können allgemeine Bestimmungen sein (z.B. gesetzliche Vorgaben).

➢ Weitere Eingabegrößen ergeben sich aus *organisatorischen Rahmenbedingungen*, denen der Prozess unterliegt (Zeitrahmen für Durchführung, zur Verfügung stehende Arbeitsmittel etc.).

Aus diesen Eingaben ergeben sich die im Prozess durchzuführenden Aktivitäten, die Ausgabeinformationen erzeugen bzw. Systemzustandsänderungen herbeiführen. Weiterhin wird die Planbarkeit dadurch bestimmt, ob die Eingaben zur Prozesslaufzeit *veränderlich* sind. Bei der Regulierung von KFZ-Schadensfällen ist eine relevante Eingabe beispielsweise, ob Personenschäden vorliegen. Mitunter werden Personenschäden aber erst während der Prozessdurchführung gemeldet. Bei Entwicklungsprozessen kommt es häufig vor, dass Anforderungen an das zu entwickelnde Produkt zur Prozessdurchführung fortwährend geändert werden.

Die Planbarkeit von Prozessen wird weiter dadurch verringert, dass Ergebnisse *fehlerhaft* oder *unvollständig* sind, wodurch die betreffenden Aktivitäten wiederholt werden müssen. Die Wahrscheinlichkeit für Fehler hängt u.a. wiederum davon ab, wie *kompliziert* ein Prozess und seine Produkte sind.

Die Planbarkeit ist in *unterschiedlichen Prozesstypen* unterschiedlich ausgeprägt. So sind beispielsweise Forschungsprozesse i.A. hochgradig dynamisch, weil die Eingaben hierfür naturgemäß hochgradig veränderlich sind, da a priori nicht oder nur sehr grob festgelegt werden kann, welche Forschungsergebnisse zu erzielen sind. Ähnliches gilt für Entwicklungsprozesse und Geschäftsprozesse, wenn auch in abgeschwächter Form. Insbesondere Entwicklungsprozesse sind hochgradig kompliziert, da hier umfangreiche Ausgaben erzeugt werden müssen, beispielsweise Auslegungen für mehrere tausend Ventile in chemischen Anlagen. Materialtransformierende Produktionsprozesse (vgl. Abbildung 1.1) hingegen sind zwar mitunter kompliziert, beispielsweise die Montage eines PKWs, Änderungen in den Eingaben werden aber nur selten zugelassen bzw. schöpfen sich aus einer vorher bekannten Menge (z.B. verfügbare Ausstattung). Die Planbarkeit in solchen Prozessem ist daher in hohem Maße gegeben.

Beispiel 1.4 (Dynamik in Geschäftsprozessen) Angenommen, die Regulierung eines Schadens sei soweit vorangeschritten, wie in Abbildung 1.4 dargestellt. Während der Prüfung zum Grunde wurde festgestellt, dass die Schadenshöhe nicht ohne ein Gutachten ermittelt werden kann. Diese zusätzliche Anforderung an den Prozess ist durch das Prozessinstanzmodell in Abbildung 1.4 nicht abgedeckt. Die Auslassung eines Gutachtens für den konkreten Prozess führt jedoch zu einer wahrscheinlich fehlerhaften Berechnung der Schadenhöhe.

Das Beispiel unterstreicht, dass die Dynamik in Geschäftsprozessen bei der Prozessdurchführung auf irgendeine Art behandelt werden muss, um Fehler wie den im Beispiel zu vermeiden.

1.2 Behandlung der Dynamik in Geschäftsprozessen

Die Dynamik in Prozessen lässt sich auf mehrere Arten behandeln. Im Folgenden werden nahe liegende Lösungsansätze erörtert, die zum Teil heutzutage gängige Praxis sind.

1.2.1 Gängige, naive Ansätze

Gängige Praxis sind folgende Ansätze: Das Vorbeiarbeiten am Prozessmanagementsystem bzw. der Neubeginn einer Prozessinstanz im Prozessmanagementsystem.

Offline-Arbeit

Das gängigste Verfahren, bei der Durchführung eines Prozesses mit einem unzureichenden Prozessdefinitionsmodell umzugehen, ist, am betreffenden Prozessmanagementsystem *vorbeizuarbeiten* [AB02, Kapitel 1]. Damit ist gemeint, dass der Prozess ohne Einbeziehung des Prozessmanagementsystems vorangetrieben wird. Die *Koordination* der beteiligten Ressourcen geschieht hierbei durch die Prozessbeteiligten selbst mit *herkömmlichen Mitteln*, also beispielsweise per Telefon oder E-Mail. Ebenso werden technischen Ressourcen im Kontext einer Prozessdurchführung manuell aufgerufen und verwendet ohne explizite Zuordnung zu einem bestimmten Prozess.

Die *Nachteile* des Vorbeiarbeitens am Prozessmanagementsystem sind so offenkundig wie zahlreich.

Erhöhte Fehleranfälligkeit Wenn die Steuerung eines Prozesses nicht mehr durch das Prozessmanagementsystem unterstützt wird, steigt die Gefahr, dass wichtige Aktivitäten im Prozess vergessen werden.

Erhöhter Arbeitsaufwand In einem Prozess, in dem mehrere Beteiligte eingebunden sind, verursacht die Übergabe prozessrelevanter Daten stets einen gewissen Aufwand. Dieser Aufwand kann nicht durch ein Prozessmanagementsystem gemindert werden, wenn an diesem vorbei gearbeitet wird.

Verringerte Nachvollziehbarkeit Ein gängige Funktionalität von Prozessmanagementsystemen ist die automatische Buchführung während einer Prozessdurchführung. Im Versicherungsbereich sind diese Daten in Bezug auf betriebswirtschaftliche Verbesserungen durch Engpassidentifizierung wichtig aber auch für die Revisionssicherheit. Wichtige Eckdaten einer Prozesshistorie, z.B. wann welche Aktivität durchgeführt wurde, können normalerweise bestimmten Dateien oder Datenbanken des Prozessmanagementsystems entnommen und automatisiert aufbereitet werden. Wird am Prozessmanagementsystem vorbei gearbeitet, so sind diese Daten nicht vorhanden oder geben zumindest ein unzutreffendes Bild der Prozesshistorie wieder.

Abbruch und Neubeginn

Ein weiteres gängiges Vorgehen ist der Abbruch einer laufenden Prozessinstanz und der anschließende Neubeginn gemäß einem anderen, passenden Prozessdefinitionsmodell. Auch hier ergeben sich mehrere Nachteile:

Geringere Nachvollziehbarkeit Abbruch und Neubeginn bedeutet nicht, dass ein Prozess letztlich nicht vollständig durch das Prozessmanagementsystem unterstützt wird. Jedoch leidet die Nachvollziehbarkeit einer Prozessdurchführung, wenn mehrere zusammenhangslose Prozesshistorien von teils abgebrochenen Prozessinstanzen einem einzigen Prozess zugeordnet werden müssen.

Erhöhter Arbeitsaufwand Wenn in einer abgebrochenen Prozessinstanz bereits einige Aktivitäten ausgeführt wurden, die auch im neuen Prozessdefinitionsmodell verwendet werden, muss die gleiche Arbeit ggf. mehrfach

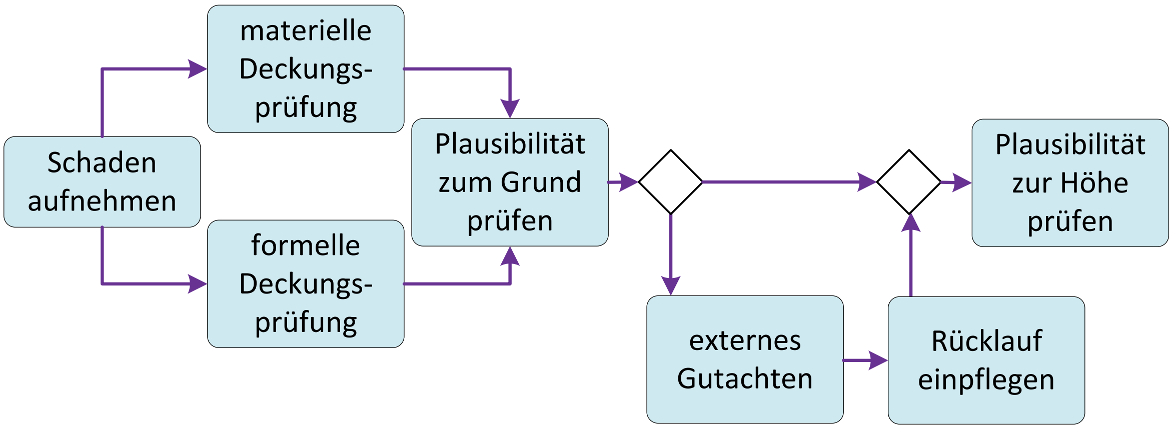


Abbildung 1.6: Erweitertes Prozessdefinitionsmodell

durchgeführt werden bzw. zumindest die Ergebnisse der Aktivitäten erneut eingegeben werden.

Fachliche Beschränkungen Eine laufende Prozessinstanz kann mitunter aus rein fachlichen Erwägungen nicht neu begonnen werden. Bei einem Versicherungsprozess ist es durchaus vertretbar, bei einer Neudurchführung interne Daten noch einmal einzugeben. Demgegenüber sollten Benachrichtigungen nach außen, beispielsweise zum Kunden, aber nicht mehrfach geschehen, um Verwirrungen zu vermeiden. Das gilt insbesondere, wenn diese Benachrichtigungen widersprüchlich sind, z.B. wenn die Regulierung eines Schadens in einer abgebrochenen Prozessinstanz positiv beschieden wird, in einer neu aufgenommenen aber negativ.

1.2.2 Nutzung der Flexibilität von Modellierungssprachen

Prozessmodellierungssprachen bieten durch Konstrukte wie Verzweigungen und Schleifen die Möglichkeit, mehrere mögliche Prozessabläufe im Vorfeld innerhalb eines Prozessdefinitionsmodells zu modellieren. Sie bieten somit eine gewisse *Flexibilität zur Modellierungszeit* eines Prozesstyps. Es ist daher naheliegend, alle *denkbaren* Abläufe von Prozessen eines bestimmten Typs in einem Prozessdefinitionsmodells festzuhalten. Man nennt solche Prozessdefinitionsmodelle häufig auch *Maximalmodelle* eines Prozesstyps.

Beispiel 1.5 (Erweitertes Prozessdefinitionsmodell)
Die Abbildung 1.6 zeigt ein gegenüber Abbildung 1.3 erweitertes Prozessdefinitionsmodell, das die im Beispiel 1.4 beschriebene Dynamiksituation

bereits abdeckt. In diesem Prozessdefinitionsmodell wird Gebrauch von einer Verzweigung gemacht – ein Konstrukt, das viele Prozessmodellierungssprachen bereitstellen. Das Prozessdefinitionsmodell lässt hierdurch offen, ob die optionalen Aktivitäten Gutachten beauftragen und Rücklauf einpflegen in einem Prozess tatsächlich durchgeführt werden.

Die beispielhafte Erweiterung ist der Verständlichkeit halber nicht sehr umfangreich. Ein typisches Maximalmodell würde gleichzeitig noch viel mehr Fälle abdecken. Jedoch sprechen mehrere Nachteile gegen die Verwendung von Maximalmodellen.

Verringerte Wartbarkeit Maximalmodelle sind sehr komplex, d.h. beinhalten eine Vielzahl von Aktivitäten, Entscheidungen und Iterationen. Die Wartbarkeit von Maximalmodellen ist daher gering. Revisionen solcher Modelle sind daher aufwändig und teuer.

Unvollständigkeit Der Begriff Maximalmodell ist insofern irreführend, als dass er auf der Annahme beruht, man könne sämtliche mögliche Prozesse a-priori planen und in einem Prozessdefinitionsmodell modellieren. Die Praxis lehrt jedoch, dass immer wieder Situationen auftreten, die im Vorfeld nicht bedacht wurden und sich demnach auch nicht in dem jeweiligen Prozessdefinitionsmodell wiederfinden [MS03, RD98]. Prozessdefinitionsmodelle sind also immer prinzipiell unvollständig. Vor diesem Hintergrund ist der Begriff *Pseudo-Maximalmodell* für ein als Maximalmodell gemeintes Prozessdefinitionsmodell angebrachter.

1.2.3 Dynamische Änderungen in Prozessinstanzmodellen

Zur Vermeidung von Maximalmodellen können Sonderfälle aus Prozessmodellen so lange herausgehalten werden, bis der jeweilige Sonderfall während der Prozessdurchführung eintritt.

Beispiel 1.6 (Geändertes Prozessinstanzmodell) In Abbildung 1.7 wurde das Prozessinstanzmodell gegenüber dem aus Abbildung 1.4 so geändert, dass nun mandatorisch zwei zusätzliche Aktivitäten nach Plausibilität zum Grunde prüfen durchgeführt werden.

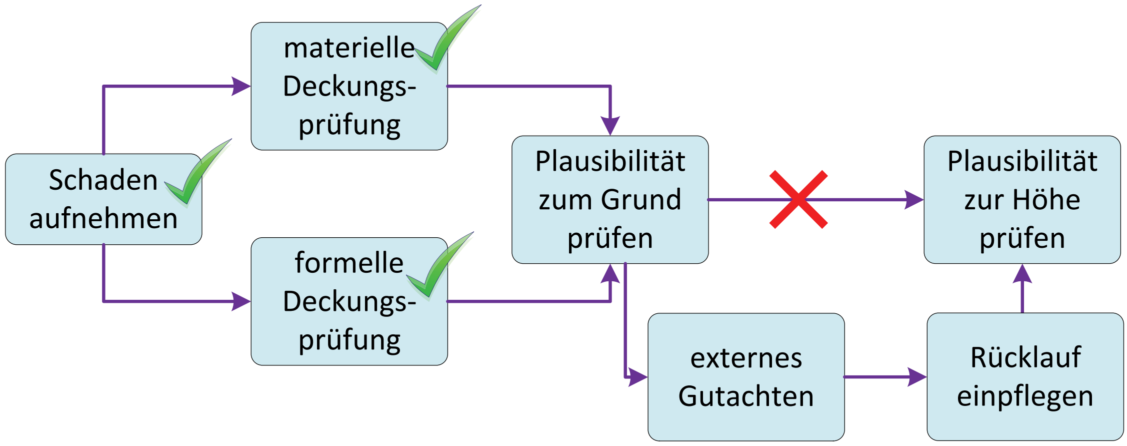


Abbildung 1.7: Dynamisch geändertes Prozessinstanzmodell

Im Unterschied zum Maximalmodellansatz wird die bestehende Kontrollflussdefinition nicht über zusätzliche Konstrukte wie Verzweigungen erweitert, sondern abgeändert, so dass die ursprüngliche Durchführung beabsichtigterweise nicht mehr möglich ist. Außerdem ist das Ziel der Änderung ein Prozessinstanzmodell statt eines Prozessdefinitionsmodells. Der Grund hierfür ist, dass ein Prozessbeteiligter zur Prozesslaufzeit i.A. nur am geänderten Ablauf des aktuellen Prozesses interessiert ist, jedoch nicht daran, andere denkbare Abläufe ebenfalls zu modellieren. Der Ablauf in prozessspezifischen Prozessinstanzmodellen wird also geändert und nicht etwa der im Prozessdefinitionsmodell modellierte Prozesstyp erweitert.

1.2.4 Schlussfolgerungen

Versicherungsprozesse als Geschäftsprozesse werden dann durch ein Prozessmanagementsystem angemessen unterstützt, wenn die gängigsten Abläufe eines Prozesstyps durch Ausnutzung der Flexibilität der Modellierungssprache vorab im Prozessdefinitionsmodell modelliert werden können *und* nicht im Prozessdefinitionsmodell vorhergesehene Abläufe durch Anpassung des jeweiligen Prozessinstanzmodells zur Prozesslaufzeit nachgetragen werden können. Der Verzicht auf Flexibilität zur Modellierungszeit würde bedeuten, dass beispielsweise gängige Ablaufalternativen nur durch fortwährende Anpassung des jeweiligen Prozessinstanzmodells realisiert werden können. Dieser Ansatz ist bei hochdynamischen Entwicklungsprozessen sinnvoll und wurde in Vorarbeiten verfolgt [Kra98, Sch02]. Standardabläufe in Geschäftsprozessen und insbesondere Versicherungsprozesse müssen jedoch kostengünstig durchgeführt werden. Die Notwendigkeit der fortwährenden Anpassung stünde dem im Wege. Der gänzliche Verzicht auf Änderbarkeit von Prozessinstanzmodellen zur Prozesslaufzeit ist jedoch ebenfalls nicht sinnvoll aus den in Unterabschnitt 1.2.2 genannten Gründen. In dieser Arbeit wird daher ein hybrider Ansatz verfolgt: Einerseits wird eine flexible Modellierungssprache eingesetzt, die insbesondere die Modellierung von Verzweigungen für Alternativen und Schleifen für Iterationen erlaubt; andererseits können Prozessinstanzmodelle zur Prozesslaufzeit geändert werden.

1.3 Anwendungsszenario Leitungswasserschadenregulierung

Im Folgenden wird der Prozesstyp "Regulierung eines Leitungswasserschadens" anhand eines entsprechenden Prozessdefinitionsmodells und aus Sicht eines Versicherers erläutert (s. Abbildung 1.8). Ziel dieses Abschnitts ist, typische Formen von Dynamik beispielhaft zu beschreiben. Daraus werden Anforderungen an eine angemessene informatische Werkzeugunterstützung für diese Art von Geschäftsprozessen abgeleitet. Die konzeptuelle und prototypische Lösung dieser Anforderungen wird detailliert in den nachfolgenden Kapiteln beschrieben.

Das Beispiel in Abbildung 1.8 ist stark vereinfacht. Gängige Aktivitäten – beispielsweise Regress-Aktivitäten – sind nicht modelliert, obwohl diese in Schadenregulierungen durchaus üblich sind.

1.3.1 Standardabläufe in der Regulierung

Ein Schadenregulierungsprozess wird für gewöhnlich in mehreren Phasen durchgeführt.

Vorbereitungsphase

In der Vorbereitungsphase werden Akten zum Schaden angelegt (nicht dargestellt). Anschließend finden Vorprüfungen statt, so genannte Deckungsprüfungen. Diese basieren im Wesentlichen auf den Angaben zum Schaden laut der anfänglichen Schadensmeldung und dem zugehörigen Vertrag.

Man unterscheidet zwischen formeller und materieller Deckungsprüfung [FW97, Seite 113]. Bei der *formellen Deckungsprüfung* (c. payment) wird geprüft, ob ein Versicherungsschutz unter formellen Gesichtspunkten im fraglichen Zeitraum überhaupt bestand. Besteht im juristischen Sinne kein Versicherungsvertrag, z.B. aufgrund nicht geleisteter Unterschriften, oder sind vom Versicherungsnehmer Beiträge nicht bezahlt worden, so kann hier bereits die Regulierung des Schadens negativ beschieden werden.

Bei der *materiellen Deckungsprüfung* (c. coverage) wird geprüft, ob für den betreffenden Schaden bestimmte Risikoausschlüsse geltend gemacht werden

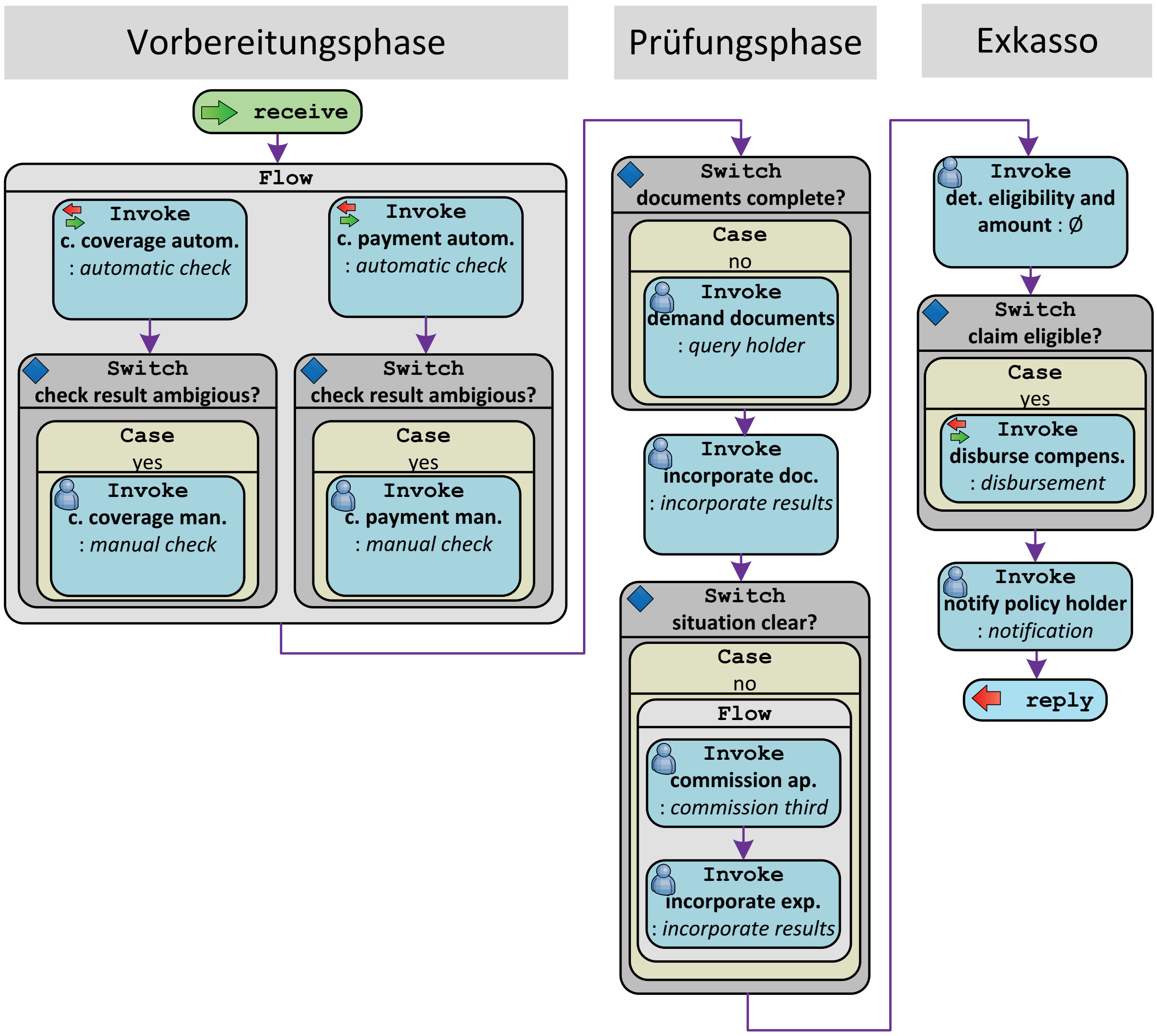


Abbildung 1.8: Beispiel eines Schadensregulierungsprozesses

können. Im Allgemeinen ausgeschlossen von der Versicherung sind beispielsweise Leitungswasserschäden, die sich wegen Bau- oder Umbaumaßnahmen in nicht bezugsfertigen bzw. bewohnbaren Gebäuden ereigneten.

Des Weiteren wird hier geprüft, ob Obliegenheiten seitens des Versicherungsnehmers verletzt wurden. Die Einhaltung von Obliegenheiten ist zwar keine Rechtspflicht des Versicherungsnehmers, ist aber zwingend erforderlich für die Erhaltung des Anspruchs aus dem Versicherungsvertrag [FW97, Seite 340]. Beispielsweise obliegt es dem Versicherungsnehmer generell, dem Versicherer Schäden unverzüglich zu melden. Eine speziell für Leitungswasserschäden geltende Obliegenheit ist, dass der Versicherungsnehmer versicherte Objekte ausreichend beheizt, um Leitungsschäden durch Frost zu verhindern.

Im Prozessdefinitionsmodell von Abbildung 1.8 sind die Vorprüfungen so modelliert, dass die formelle parallel zur materiellen Deckungsprüfung durchgeführt werden kann. Dies wird dadurch erreicht, indem die betreffenden Aktivitäten in einem so genannten `Flow`-Element enthalten sind. Die Parallelität wird jedoch insoweit eingeschränkt, dass die automatischen vor den jeweiligen manuellen Prüfungen durchgeführt werden müssen, ausgedrückt durch gerichtete Kanten (`Links`) im Modell. Die manuellen Prüfungen werden nur optional durchgeführt und zwar nur dann, wenn die automatischen zu keinem eindeutigen Ergebnis führen. Die Modellierung spiegelt das jeweils durch eine Verzweigung (`Switch`) wider, deren Zweig (`Case`) jeweils übersprungen wird, wenn die Ergebnisse der automatischen Prüfungen nicht mehrdeutig (ambigious) sind. In Kapitel 2 wird näher auf die Syntax und Semantik von Prozessdefinitionsmodellen eingegangen.

Prüfungsphase

Ergeben die formelle und materielle Deckungsprüfung, dass der Anspruch des Versicherungsnehmers laut Schadensmeldung prinzipiell gerechtfertigt ist, so werden Belege geprüft, die die Schadensmeldung stützen. Solche Belege sind typischerweise ausgefüllte Formulare, in denen der Versicherungsnehmer Stellung zum Ausmaß, Hergang etc. nehmen muss, sowie Fotografien vom Schaden.

Ist die anfängliche Schadensmeldung telefonisch erfolgt (synchrone Kommunikation), so wird der Versicherungsnehmer i.A. zum Beibringen dieser Belege aufgefordert, so dass diese nun vorliegen. Ging die Schadensmeldung in schriftlicher Form per Brief, Fax oder E-Mail ein (asynchrone Kommunikation), wird der Versicherungsnehmer normalerweise erst an dieser Stelle aufgefordert, die Belege nachzuliefern (demand documents), die eingepflegt werden müssen (incorporate doc.).

Wenn sich aus den vorhandenen Belegen keine eindeutige Sachlage ergibt, wird ein Sachverständiger beauftragt, den Schaden zu begutachten (commission ap.). Der Rücklauf des Gutachtens muss ebenfalls eingepflegt werden (incorporate exp.). Aus den Belegen und dem eventuellen Gutachten lässt sich in det. eligibility and amount ableiten, ob der Anspruch generell gerechtfertigt ist und in welcher Höhe er erstattet werden muss.

Exkasso

Stellt der Sachbearbeiter fest, dass die angegebenen Gründe plausibel sind, ermittelt er anschließend anhand vorhandener Belege die Höhe der zu zahlenden Entschädigung und veranlasst daraufhin die Auszahlung des Schadens (disburse compensation).

Zuletzt wird der Versicherungsnehmer über die Bewilligung der Entschädigung bzw. deren Ablehnung vom Sachbearbeiter informiert (notify policy holder).

1.3.2 Dynamik in der Regulierung

In den meisten Fällen kann ein Prozess zur Regulierung von Leitungswasserschäden so, wie er im Prozessdefinitionsmodell in Abbildung 1.8 modelliert ist, durchgeführt werden. Im Folgenden sollen Beispiele für Dynamik im Regulierungsprozess von Leitungswasserschäden erläutert werden, also Abläufe, die vom Prozessdefinitionsmodell aus Abbildung 1.8 nicht abgedeckt werden.

Ungeplante, zusätzlich auszuführende Aktivitäten

Beispiel 1.7 (Zusätzliches determine payee) Im Prozessdefinitionsmodell aus Abbildung 1.8 wird implizit davon ausgegangen, dass der Versicherungsnehmer auch der Zahlungsempfänger ist. Da in heutigen Versicherungsprozessen Geld bargeldlos transferiert wird, setzt das das Vorhandensein einer gültigen Bankverbindung beim Versicherungsnehmer voraus.

In manchen Fällen kann es aber sein, dass der Versicherungsnehmer über keine bzw. keine gültige Bankverbindung verfügt. In solchen Fällen muss eine zusätzliche Aktivität im Prozess durchgeführt werden, in der eine gültige Bankverbindung ermittelt wird. In Abbildung 1.9, die ein dynamisch geändertes Prozessinstanzmodell zeigt, ist dies die Aktivität determine payee. Das Prozessinstanzmodell modelliert einen Prozess, in dem aktuell die Aktivität incorporate documents durchgeführt wird. Die strichlierten Pfeile zeigen die tatsächliche bisherige Ausführungsreihenfolge der Aktivitäten an.

Zwei Zeitpunkte sind für unvorhergesehene, zusätzlich auszuführende Aktivitäten wichtig: zum einen der Zeitpunkt, an dem erkannt wird, dass Aktivitäten im aktuellen Prozess fehlen und zum anderen der Zeitpunkt, an denen die Aktivitäten im Prozess durchgeführt werden sollen.

Die Stelle in der Kontrollflussdefinition, in der zusätzliche Aktivitäten eingefügt werden, muss unabhängig vom Zeitpunkt sein, an dem sie eingefügt werden. Im Prozessinstanzmodell von Abbildung 1.9 ist der Kontrollfluss bis incorporate documents fortgeschritten, erkennbar an den Zustandsbeschriftungen Finished innerhalb der Aktivitäten. So kann determine payee

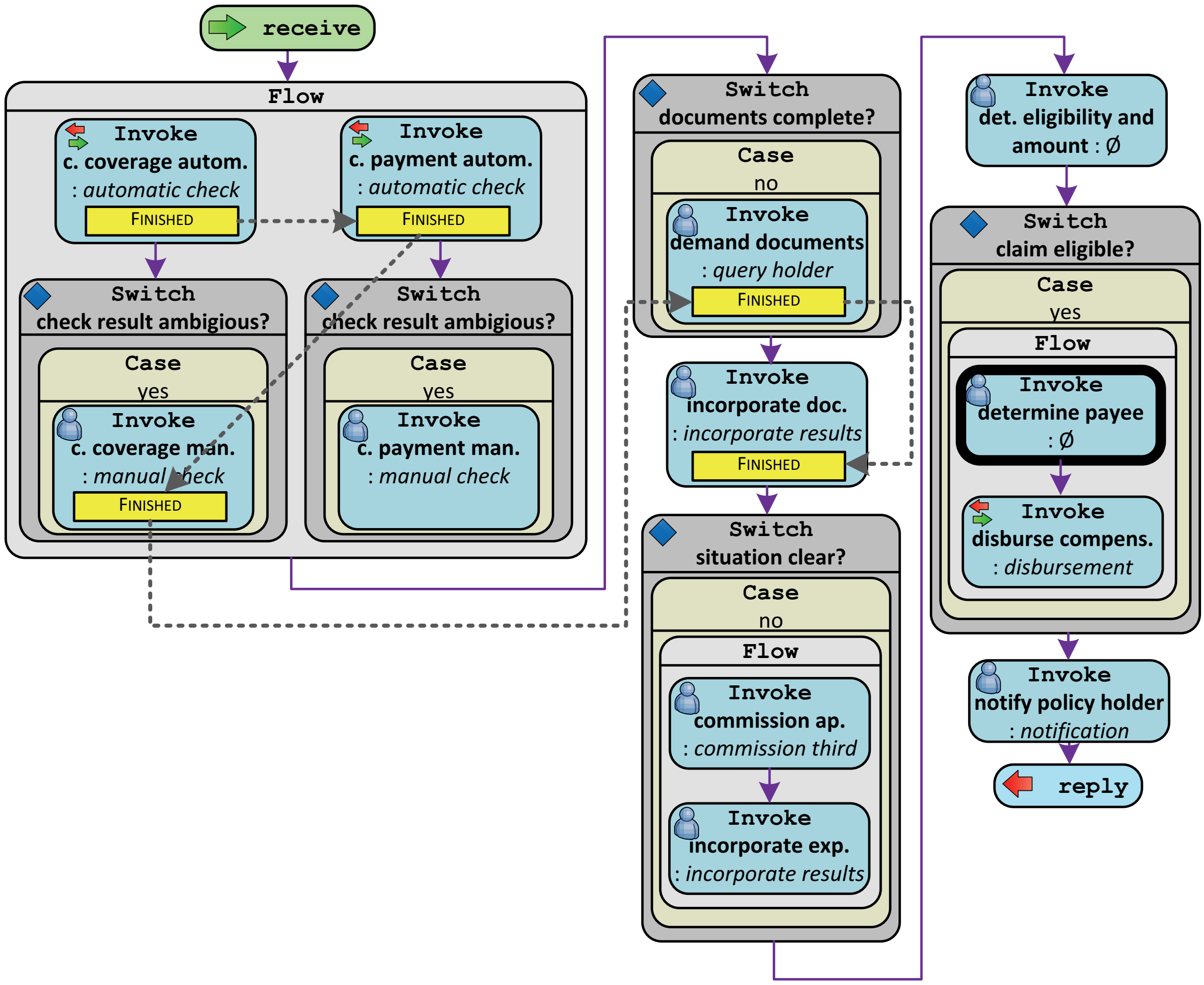


Abbildung 1.9: Beispiel einer nachträglich eingefügten Aktivität

vor disburse compensation eingefügt werden, obwohl der Kontrollfluss noch einige Aktivitäten vor der Einfügestelle steht.

Diese Unabhängigkeit ist wichtig und muss durch ein dynamisches Prozessmanagementsystem unterstützt werden. Wären mögliche Einfügestellen prinzipiell abhängig von der aktuellen Kontrollflussposition, würde dies eine starke Einschränkung in Bezug auf die praktische Nutzbarkeit des betreffenden dynamischen Prozessmanagementsystems darstellen. Angenommen, mögliche Einfügestellen wären nur solche, die in der Kontrollflussdefinition unmittelbar auf die Aktivität folgen, an der sich der Kontrollfluss aktuell befindet. Weiterhin werde angenommen, der Kontrollfluss befinde sich aktuell bei der Aktivität incorporate documents, d.h. bei der Durchführung dieser Aktivität werde die Notwendigkeit der zusätzlichen Aktivität festgestellt. Dann gibt es folgende Möglichkeiten, auf diese Einschränkung zu reagieren:

Verzögertes Einfügen Das Einfügen der zusätzlichen Aktivität kann soweit hinausgezögert werden, bis der Kontrollfluss bis zu der Aktivität fortgeschritten ist, die unmittelbar vor der beabsichtigten Einfügestelle liegt. Dieses Vorgehen verlangt natürlich, dass die Information für das verzögerte Einfügen in irgendeiner Form gespeichert wird bzw. von einem Sachbearbeiter gemerkt wird. Ersteres stellt lediglich eine etwas umständliche Realisierung des direkten Einfügens dar, letzteres birgt die Gefahr, dass das verzögerte Einfügen schlichtweg vergessen wird.

Verfrühte Durchführung Eine andere Möglichkeit ist, die einzufügende Aktivität direkt hinter der aktuellen Kontrollflussposition zu platzieren. In dem Beispiel würde also determine payee direkt hinter incorporate documents eingefügt. Der Nachteil hiervon ist, dass zum Zeitpunkt der Durchführung von incorporate documents noch nicht feststeht, ob eine Bankverbindung überhaupt benötigt wird. Gegebenenfalls fällt das Gutachten so aus, dass der Schaden nicht versichert ist, die Aktivitäten in claim eligible also nicht durchgeführt werden dürfen. Eine verfrühte Durchführung von determine payee könnte sich also im Nachhinein als überflüssig herausstellen.

Der in dieser Arbeit entwickelte Ansatz erlaubt, zu beliebigen Zeitpunkten an prinzipiell beliebigen Stellen in der Kontrollflussdefinition des Prozesses Aktivitäten nachträglich einzufügen, sofern bestimmte Bedingungen dem nicht widersprechen (vgl. Unterabschnitt 1.3.3). Somit ist es unerheblich, ob das Fehlen der Bankverbindung bereits früh bei der Prozessdurchführung festgestellt wird oder erst kurz vor der Auszahlung der Entschädigung.

Überflüssige, auszulassende Aktivitäten

Zuvor wurde eine Situation beschrieben, in der gewisse Aktivitäten für einen bestimmten Prozess notwendig, im Prozessdefinitionsmodell aber nicht vorgesehen sind und daher nachträglich eingefügt werden müssen. Komplementär hierzu ist die Situation, dass die Durchführung bestimmter Aktivitäten für einen bestimmten Prozess nicht sinnvoll ist.

Beispiel 1.8 (Überflüssiges incorporate documents) Für die Regulierung von Leitungswasserschäden sind beispielsweise Situationen vorstellbar, in denen der zu regulierende Schaden so gering ist, dass keine Belege vonnöten sind. Das bedeutet, dass die formelle als auch materielle Deckungsprüfung durchgeführt würden, da auch in diesem Fall keine Schadenauszahlung stattfinden darf, wenn der Schaden formell oder materiell ungedeckt ist. Jedoch würde der Prozessteil in der Mitte von Abbildung 1.8, in dem die Vollständigkeit und das Einpflegen der Belege sichergestellt

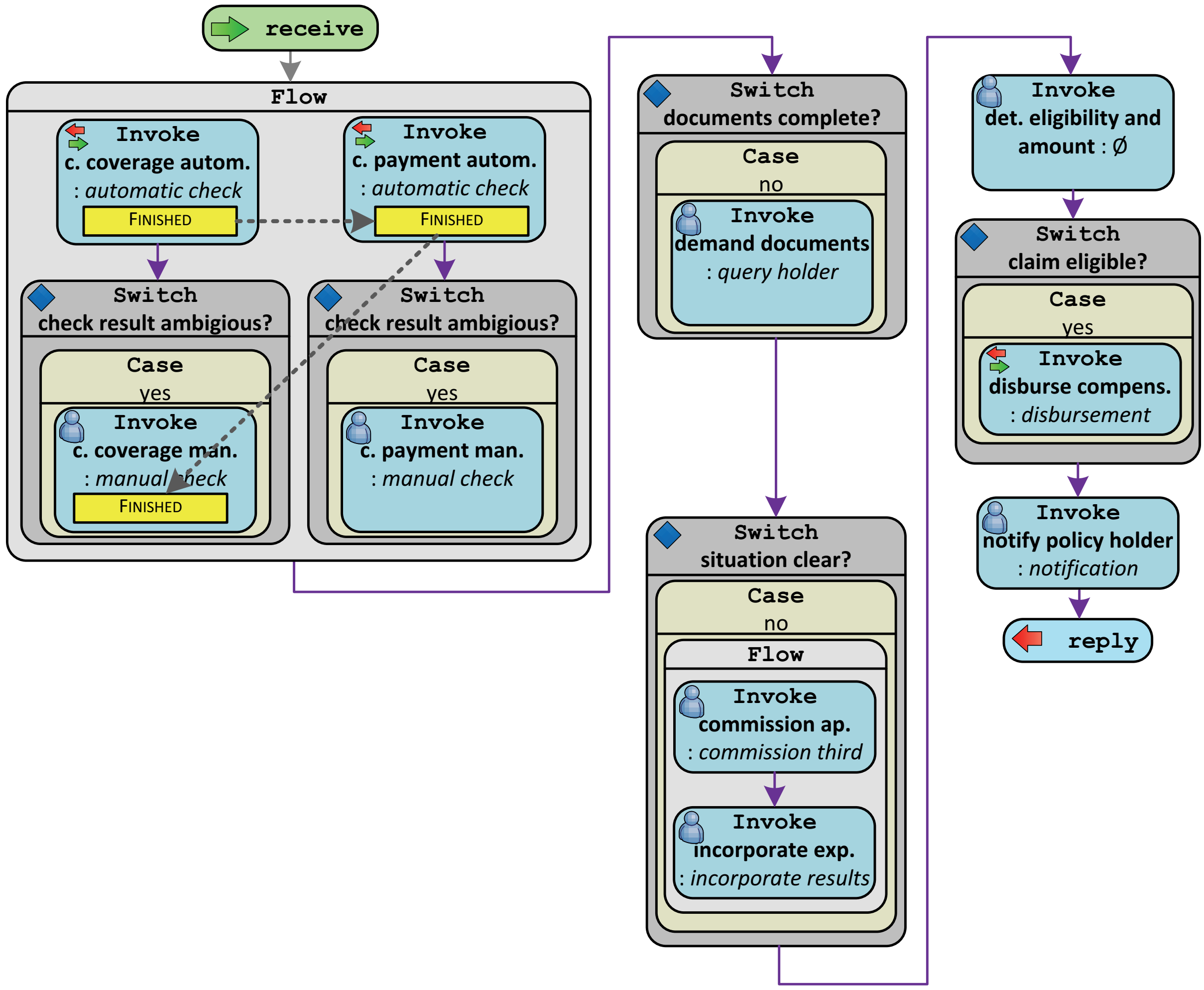


Abbildung 1.10: Beispiel einer nachträglich entfernten Aktivität

wird, im Wesentlichen übersprungen. Das heißt, dass in documents complete trivialerweise die Vollständigkeit der Belege festgestellt würde und somit die in einer Verzweigung befindliche Aktivität demand documents nicht durchgeführt wurde. Das Problem ist nun, dass die Aktivität incorporate documents in jedem Fall durchgeführt wird. Das ist nur für solche Prozesse sinnvoll, für die Belege notwendig sind. Abbildung 1.10 zeigt ein dynamisch geändertes Prozessinstanzmodell, in dem dieses Problem nicht mehr auftritt. Hier wurde die Aktivität incorporate documents dynamisch gelöscht.

Das Problem der Durchführung von überflüssigen Aktivitäten, bezogen auf einen bestimmten Prozess, ist nicht unmittelbar einsichtig. Solche Aktivitäten lassen sich durchaus einfach abschließen, ohne dass sie tatsächlich nennenswerte Aufwände verursachen. Bei der Vielzahl von Prozessen, in die ein Versicherungssachbearbeiter im Allgemeinen involviert ist, führen überflüssige Aktivitäten jedoch zumindest zu Verwirrung. Es bedarf also einer Funktion, überflüssige Aktivitäten aus laufenden Prozessen zu entfernen.

Auch für das Entfernen von Aktivitäten ist die prinzipielle Unabhängigkeit der aktuellen Kontrollflussposition von der zu entfernenden Aktivität bzw. den zu entfernenden Aktivitäten wichtig. Wird das Entfernen verzögert, bis der Kontrollfluss die zu entfernende Aktivität erreicht hat und die Aktivität faktisch entfernt, indem sie ohne Eingaben abgeschlossen wird, führt das zu den oben angesprochenen Problemen.

In dieser Arbeit werden dynamische Löschungen daher so realisiert, dass sie prinzipiell unabhängig von der Kontrollflussposition stattfinden, jedoch vorbehaltlich der Fälle, in denen gewichtige technische oder fachliche Gründe gegen eine dynamische Löschung sprechen (vgl. Unterabschnitt 1.3.3).

Fehlerhaft durchgeführte, zu wiederholende Aktivitäten

Eine typische Situation in Geschäftsprozessen ist, dass bestimmte Aktivitäten fehlerhaft durchgeführt wurden.

Beispiel 1.9 (Rücksprung) Es ist beispielsweise denkbar, dass ein Prozess gemäß Abbildung 1.8 bereits bis zu determine eligibility and amount fortgeschritten ist. Der Sachbearbeiter, der diese Aktivität durchführt bemerkt nun, dass zur Beurteilung der Sachlage wichtige Belege fehlen. Dabei kann es sich beispielsweise um Quittungen bereits ersetzter Einrichtungsgegenstände handeln.

Die fehlerhafte Durchführung lässt sich nur adäquat mittels eines Rücksprungs zu der Aktivität beheben, bei der der Fehler entstanden ist. In dem Beispiel heißt das, dass die aktuelle Kontrollflussposition von determine eligibility and amount zu documents complete zurückgesetzt werden muss (s. Abbildung 1.11)

Relevante Aktivitäten bei einem Rücksprung sind insbesondere die Aktivitäten, die zwischen der Aktivität liegen, zu der zurückgesprungen werden soll, und der Aktivität, von der aus zurückgesprungen wird. In Beispiel 1.9 sind das die Aktivitäten demand documents, incorporate documents, commission appraiser und determine eligibility and amount. Es ist wichtig, dass die Effekte dieser Aktivitäten erhalten bleiben. Ein Rücksprung ist also nicht gleichbedeutend mit einem aus dem Datenbankbereich bekannten Rollback. Der Grund hierfür ist, dass die in diesen Aktivitäten geleistete Arbeit im

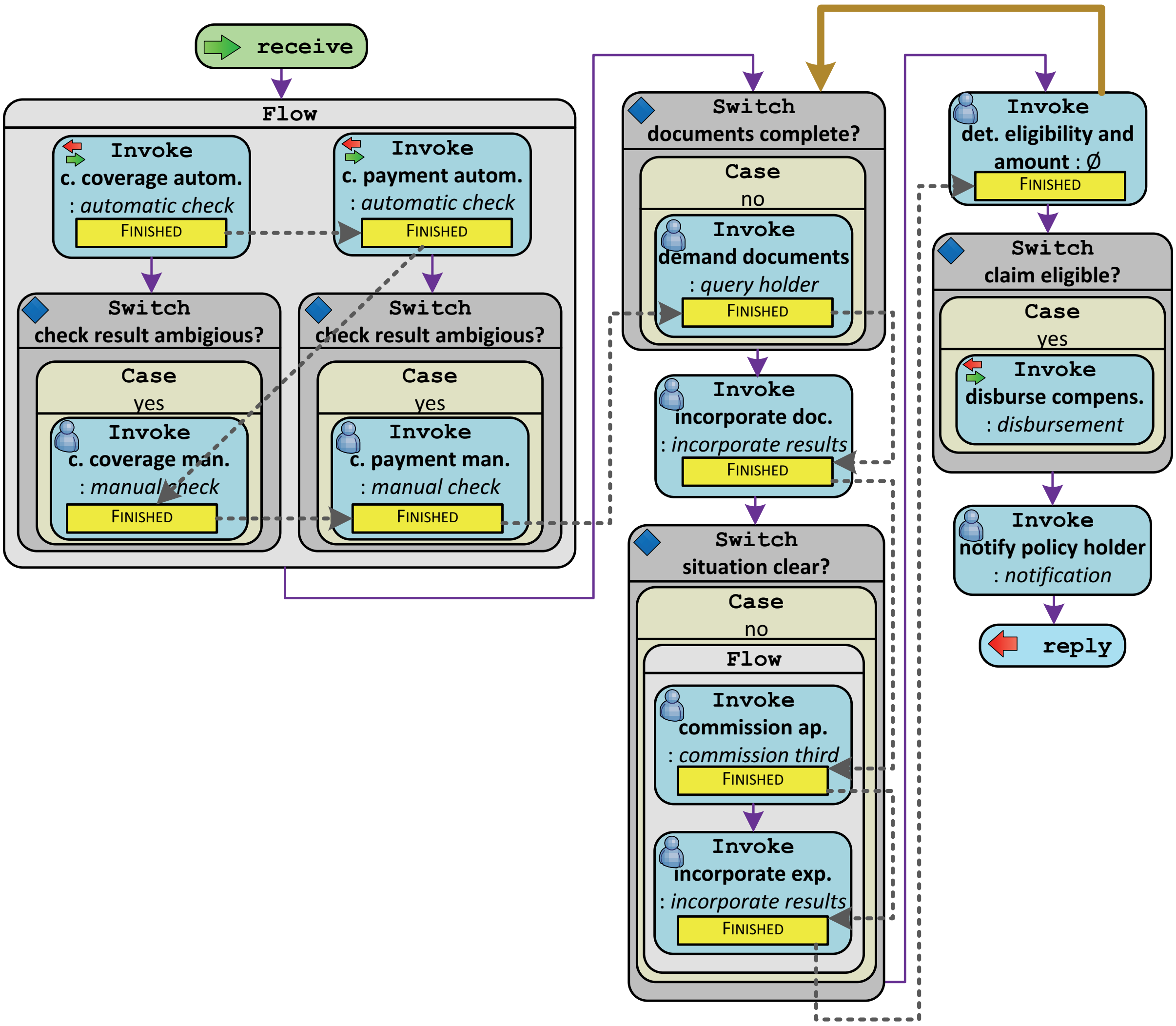


Abbildung 1.11: Beispiel eines nachträglich eingefügten Rücksprungs

Allgemeinen nur in Teilen Fehler bzw. Folgefehler aufweist. Es ist beispielsweise nicht einzusehen, weshalb bei der ersten Durchführung von incorporate documents eingepflegte Belege gelöscht werden sollten. Viel mehr sollten die eingepflegten Belege erhalten bleiben, jedoch um die zusätzlichen, vormals nicht vorhandenen, ergänzt werden.

1.3.3 Bedingungen an Regulierungsprozesse

In Unterabschnitt 1.3.2 wurden mögliche Prozessänderungen erläutert. Prozessänderungen unterliegen bestimmten Rahmenbedingungen. Dies trifft auf Änderungen zur Definitionszeit eines Prozesses zu, aber auch auf Änderungen zur Laufzeit eines Prozesses. Die Verletzung dieser Rahmenbedingungen führt im Allgemeinen zu Problemen.

Die Probleme können verschiedener Art sein. Obwohl die Grenze fließend ist, kann man grob zwischen technischen und fachlichen Problemen bei Prozessänderungen unterscheiden. Im Folgenden werden hierzu Beispiele aufgeführt. Die Erläuterungen nehmen Bezug auf das Prozessinstanzmodell in Abbildung 1.12, welches abgeleitet vom Prozessdefinitionsmodell aus Abbildung 1.8 ist, jedoch mehreren dynamischen Änderungen unterzogen wurde.

Technische Probleme

Unter technischen Problemen werden solche Probleme verstanden, die durch Modelländerungen – beispielsweise dynamischen Änderungen in Prozessinstanzmodellen – verursacht werden und dazu führen können, dass das betreffende Modell vom Prozessmanagementsystem nicht mehr ordnungsgemäß verarbeitet werden kann. Technische Probleme können auch unter dem Blickwinkel gesehen werden, dass sie unabhängig von dem Anwendungsbereich der Prozessmodelle, also domänenunabhängig sind, d.h. in Versicherungsprozessen genauso auftreten können wie in Geschäftsprozessen in Behörden oder Banken.

Nicht-initialisierte Daten Typische technische Problemquellen sind beispielsweise Lesezugriffe auf nicht-initialisierte Daten. In Prozessdefinitions- und -instanzmodellen tritt ein solches Problem zutage, wenn folgende Bedingungen eintreten:

- Der Kontrollfluss erreicht eine Aktivität $A1$, die eine Variable V lesend verwendet.
- Variable V ist zu diesem Zeitpunkt noch nicht initialisiert. Es gibt also keine Aktivität $A2$, die zu einem früheren Zeitpunkt als $A1$ vom Kontrollfluss erreicht wurde und die schreibend auf V zugreift.

Beispiel 1.10 (Nicht-initialisiertes eligible) Im Prozessinstanzmodell aus Abbildung 1.12 wurde eine Aktivität zur zusätzlichen Plausibilitätsprüfung (amount plausibility check) eingefügt. Diese Aktivität greift lesend (strichlierter Pfeil) auf das Datum eligible zu (strichliertes Rechteck), welches erst später im Prozess von determine eligibility and amount geschrieben wird.

Unerreichbare Aktivitäten Gerade bei dynamischen Änderungen im Prozessinstanzmodell passiert es leicht, dass Aktivitäten dynamisch an solchen

Stellen eingefügt werden, die aufgrund des Fortschrittes des Kontrollflusses gar nicht mehr erreicht werden können.

Beispiel 1.11 (Unerreichbares amount plausibility check)
Im Beispiel von Abbildung 1.12 kann die dynamisch eingefügte Aktivität amount plausibility check nicht mehr erreicht werden, da der Kontrollfluss bereits bis demand documents fortgeschritten ist.

Ein weiteres Problem im Prozessinstanzmodell von Abbildung 1.12 ist das Datum plausible, welches nur geschrieben aber nie gelesen wird. Zwar führt dieses Problem nicht dazu, dass die Verarbeitung in einem Prozessmanagementsystem daran scheitert, jedoch ist diese Situation unabhängig von fachlichen Rahmenbedingungen unerwünscht und fällt daher auch in die Gruppe technischer Probleme.

Fachliche Probleme

Änderungen in Prozessen können neben technischen auch rein fachliche Probleme erzeugen. Unter rein fachlichen Problemen sind solche Änderungen oder Situationen in Prozessmodellen zu verstehen, die zwar keine negativen Auswirkungen auf die technische Ausführbarkeit des Prozessmodells haben, jedoch gegen domänenspezifische Bestimmungen verstoßen. Fachliche Probleme werden im Folgenden wiederum beispielhaft mit Bezug auf das Prozessinstanzmodell aus Abbildung 1.12 erläutert.

Beispiel 1.12 (Rücklauf einpflegen) Kommt es bei der Regulierung von Leitungswasserschäden zur Mitarbeit Dritter, beispielsweise externer Gutachter, so können Prozessänderungen leicht fachliche Probleme nach sich ziehen. Kann anhand der Beleglage nicht eindeutig festgestellt werden, ob der Schaden reguliert werden soll oder nicht, so wird in dem Prozessinstanzmodell aus Abbildung 1.12 ein externes Gutachten in Auftrag gegeben (commission appraiser).

Die Beauftragung führt normalerweise zu einem schriftlichen, häufig noch postalisch versendeten Rücklauf. Dieser Rücklauf muss wiederum in den laufenden Prozess innerhalb der Versicherung eingebracht werden. Im Prozessinstanzmodell bedeutet das, dass es eine Aktivität incorporate expertise gibt, die den entstandenen Medienbruch behebt.

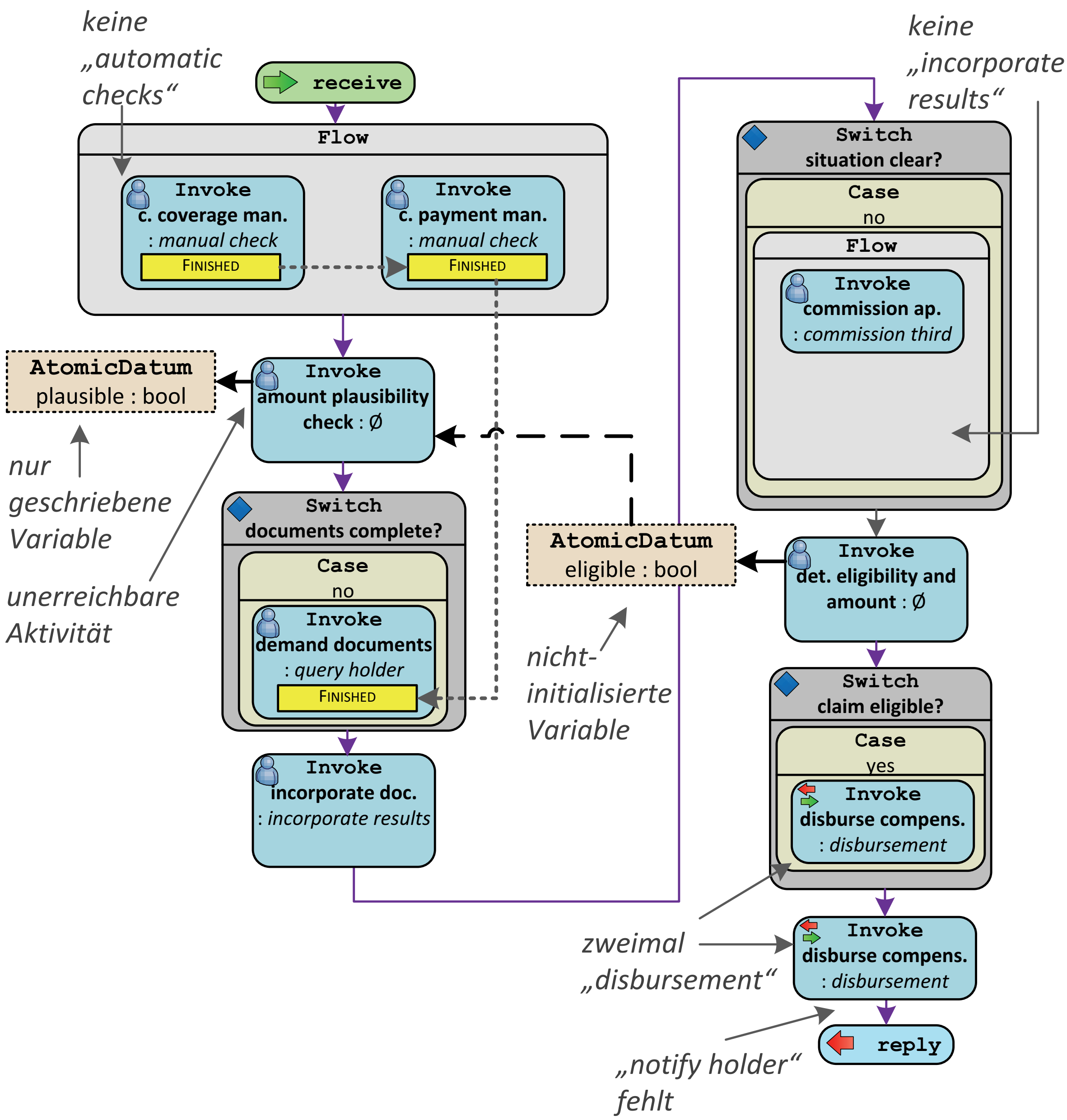


Abbildung 1.12: Prozessinstanzmodell mit problematischen Änderungen

Für den menschlichen Betrachter ist unmittelbar plausibel, dass die Aktivitäten commission appraiser und incorporate expertise zusammengehören; das heißt, wenn erstere Aktivität durchgeführt wird, sollte auch letztere durchgeführt werden und zwar in dieser Reihenfolge.

Es kann nun jedoch sein, dass ein Sachbearbeiter die Aktivität incorporate expertise versehentlich dynamisch löscht. Aus technischer Sicht ist hiergegen nichts einzuwenden.

Aus fachlicher Sicht ist diese Löschung jedoch unsinnig: Die Löschung bewirkt, dass der Rücklauf eben nicht eingepflegt wird. Hierdurch kann das externe Gutachten im weiteren Prozessverlauf keine Rolle mehr spielen. Die Aktivität commission appraiser würde durch die Löschung von incorporate expertise ebenfalls überflüssig. Beide Aktivitäten sind also nur in Kombination in einem Prozess sinnvoll.

Beispiel 1.13 (Automatische vor manueller Deckungsprüfung) Die Prüfung der Deckung ist in Schadensregulierungsprozessen üblich. Teile hiervon sind routinemäßige Tätigkeiten, beispielsweise die Überprüfung der Zahlungseingänge vom Versicherungsnehmer. Diese Tätigkeiten lassen sich daher gut automatisieren, was im Prozessdefinitionsmodell aus Abbildung 1.8 durch die Aktivitäten check coverage automatically und check payment automatically widergespiegelt wird.

Eine solche automatische Prüfung kann in Zweifelsfällen zu einem indifferenten Ergebnis kommen. In diesem Fall werden die Deckungsprüfungen manuell durchgeführt. Eine manuelle Deckungsprüfung führt in jedem Fall zu einem eindeutigen Ergebnis. Insofern ist sie der automatischen Prüfung überlegen. Trotzdem ist die automatische Prüfung sinnvoll, wenn auch nicht zwingend notwendig, da sie durchgeführt werden kann, ohne nennenswerte Prozesskosten zu verursachen. Führt die automatische Prüfung zu einem eindeutigen Ergebnis, wird die manuelle Prüfung nicht zusätzlich durchgeführt, wodurch Kosten gespart werden.

Die wesentliche Bedingung an einen Versicherungsprozess und damit auch an mögliche Prozessänderungen ist nicht, dass in jedem Fall eine automatische Deckungsprüfung durchgeführt wird. Vielmehr ist wichtig, dass eine automatische Deckungsprüfung erfolgt, wenn eine manuelle stattfindet und dass die automatische Deckungsprüfung *vor* der manuellen geschieht. Nur damit ist sie sinnvoll, da nur dann eine manuelle Prüfung überflüssig werden kann und somit Prozesskosten gesenkt werden können. Im Prozessinstanzmodell von Abbildung 1.12 fehlen die automatischen

Prüfungen gänzlich. Insoweit ist dies ein fachliches Problem.

Beispiel 1.14 (Obligatorische Aktivitäten) In den vorhergehenden beiden Beispielen sprachen fachliche Gründe gegen die Löschung bzw. Verschiebung bestimmter Aktivitäten aufgrund des Vorhandenseins bzw. den Positionen bestimmter anderer Aktivitäten. Eine schwächere Form einer Bedingung an einen Prozess ist die Forderung, dass eine bestimmte Aktivität im Verlauf eines Prozesses durchgeführt werden muss, unabhängig vom Vorhandensein bzw. von den Positionen anderer Aktivitäten.

In einem Versicherungsprozess ist es wichtig, dass ein Versicherungsnehmer benachrichtigt wird. Deshalb ist eine grundlegende Forderung, dass die Aktivität notify policy holder im Prozess durchgeführt wird. Auch hier ist es wiederum so, dass gegen eine Löschung aus technischer Sicht nichts einzuwenden wäre. Datenflussabhängigkeiten werden auch hier nicht verletzt, da notify policy holder auf keine Prozessdaten schreibend zugreift. Aus fachlicher Sicht ist es natürlich wiederum unstreitig, dass der Versicherungsnehmer benachrichtigt werden muss, unabhängig davon, ob der Schaden von der Versicherung übernommen wird oder nicht.

1.4 Neue Beiträge und Aufbau der Arbeit

Aufbauend auf den bisherigen Erläuterungen wird in diesem Abschnitt die vorliegende Arbeit von bisherigen Vorarbeiten abgegrenzt. Anschließend werden die kapitelweise behandelten Konzepte zur Unterstützung dynamischer Geschäftsprozesse als Vorausschau jeweils kurz beschrieben und wesentliche wissenschaftliche Beiträge herausgestellt.

1.4.1 Ausgangspunkt und Abgrenzung

Es liegt nahe, zur Unterstützung dynamischer Prozesse ein völlig neuartiges Prozessmanagementsystem zu konzipieren ohne Berücksichtigung bestehender Prozessmodellierungssprachen und -systeme. Diese Vorgehensweise wurde insbesondere in den Vorgängerarbeiten dieser Arbeit [Kra98, Wes99b, Sch02, Jäg03, Hel08] verfolgt. Grundlegende Konzepte der Vorgängerarbeiten konnten dabei in dieser Arbeit übernommen werden, beispielsweise die Modellierung von Prozessen auf unterschiedlichen Abstraktionsebenen.

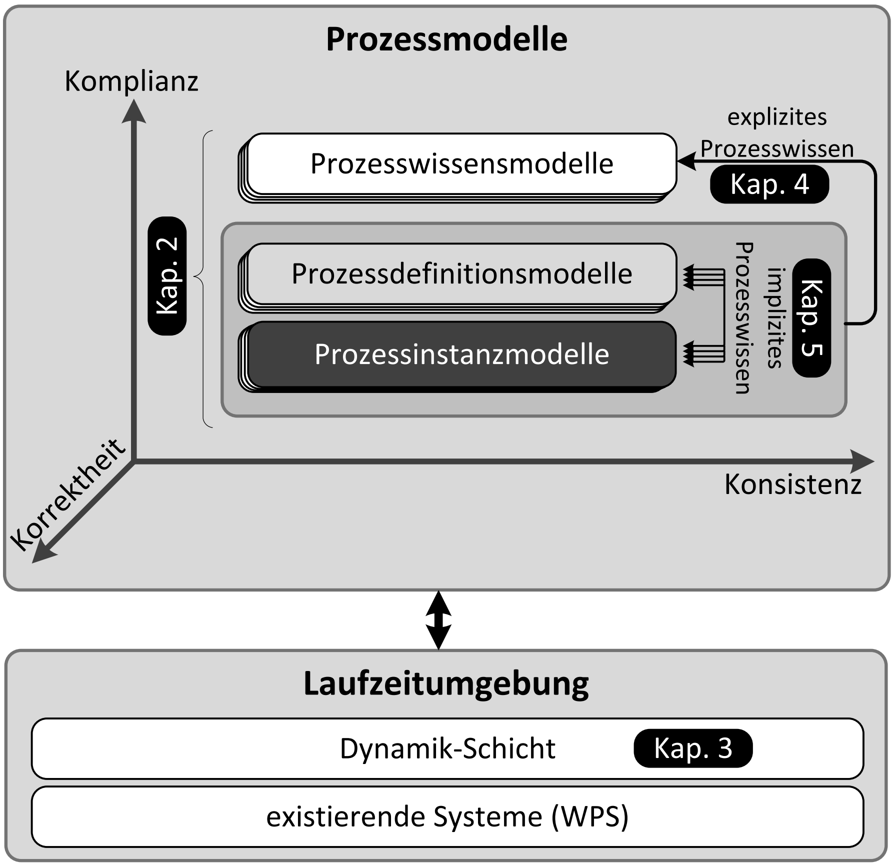


Abbildung 1.13: Aufbau der Arbeit

Die Anwendungsbereiche der Vorgängerarbeiten waren stets Prozesse im Softwareentwicklungsbereich und in der Entwicklung verfahrenstechnischer Anlangen. Diese Prozesse zeichnen sich dadurch aus, dass dynamische Änderungen zur Prozesslaufzeit normal sind. Entwicklungsprozesse werden immer nur auf einer grobgranularen Ebene modelliert und die Modelle zur Laufzeit verfeinert und erweitert. Die dynamischen Änderungen für diese Prozesse lassen sich also mit *fortwährender Evolution* charakterisieren. Geschäftsprozesse, beispielsweise im Versicherungsumfeld, sind repetitiver. Infolgedessen kann ein Geschäftsprozess durchaus so modelliert werden, dass er im Normalfall wie vorab modelliert durchgeführt werden kann. Trotzdem treten auch bei Geschäftsprozessen unvorhergesehene Situationen auf, die eine dynamische Änderung zur Laufzeit notwendig machen. Diese sind allerdings eher als *Abweichungen* von der vorhandenen Modellierung zu sehen und weniger als Evolution.

Abbildung 1.13 zeigt die in dieser Arbeit behandelten Teilgebiete und deren Zusammenhänge. Diese werden im Folgenden kurz skizziert und der jeweilige wissenschaftliche Beitrag dargestellt.

1.4.2 Systematische und formale Modellbildung

Die Modellbildung ist für die werkzeugseitige Unterstützung des Prozessmanagements unerlässlich. Modelle dienen letztendlich auch dazu, Informationen zwischen Prozessmanagementsystem und Prozessbeteiligten zu kommunizieren. In dieser Arbeit werden Prozesse auf drei unterschiedlichen, aufeinander aufbauenden Abstraktionsebenen modelliert, wie bereits in [WKH08b] skizziert. Die unterste Ebene bilden Prozessinstanzmodelle, die konkrete Prozesse widerspiegeln. Prozessdefinitionsmodelle abstrahieren gegenüber Prozessinstanzmodellen von Ausführungszuständen, modellieren also Prozesstypen. Ein weiterer Abstraktionsschritt wird in Prozesswissensmodellen vorgenommen, die von Details der Ausführung abstrahieren, beispielsweise der exakten Ausführungsreihenfolge von Aktivitäten.

In Kapitel 2 werden die unterschiedlichen Sprachen für Prozessmodelle eingeführt und insbesondere Querbezüge zwischen Prozessmodellen auf unterschiedlichem Abstraktionsniveau verdeutlicht. Hierbei wird die Syntax und Semantik der Sprachen auf formalem Weg erläutert.

Beiträge Mit der Modellierung von Prozessen auf unterschiedlichen, wohldefinierten Abstraktionsebenen überführt die Arbeit einen bewährten Ansatz in die industrielle Praxis. Die Trennung in mehrere Abstraktionsebenen ist zurückzuführen auf [NW94] und wurde bereits in Vorgängerarbeiten [Kra98, Sch02] aufgegriffen. Die Vorgängerarbeiten basieren auf grundsätzlich neuentwickelten Prozessmodellierungssprachen [HJKW96] und Systemen [JSBW00]. Ziel dieser Arbeit ist demgegenüber, die vorhandenen Konzepte so weiterzuentwickeln und anzupassen, dass bestehende Standards und existierende, kommerzielle Systeme des Projektpartners Generali Deutschland Informatik Services GmbH berücksichtigt werden. Hierbei muss insbesondere WS-BPEL (WebServices Business Process Execution Language) als Standard für die feingranulare Modellierung von Prozessdefinitionsmodellen in die neuen Konzepte miteinbezogen werden. WS-BPEL unterscheidet sich von den in Vorgängerarbeiten verwendeten DYNAMITE-Aufgabennetzen insbesondere dadurch, dass diese Sprache die Modellierung von optional und iterativ durchgeführten Aktivitäten erlaubt. Hierdurch kann eine höhere, jedoch für dynamische Prozesse nicht ausreichende Flexibilität zur Modellierungszeit erreicht werden.

1.4.3 Erweiterung eines Prozessmanagementsystems

Das Prozessmanagementsystem WebSphere Process Server verwaltet Prozessdefinitionsmodelle und davon abgeleitete Prozessinstanzen. Die Ablaufstruktur muss stets so verbleiben, wie vorab im Prozessdefinitionsmodell festgelegt.

In Kapitel 3 wird ein Ansatz erläutert, der es ermöglicht, den WebSphere Process Server so zu erweitern, dass die wichtigsten Formen dynamischer Änderungen in Prozessinstanzmodellen unterstützt werden. Dies sind (1) dynamische Einfügungen zusätzlicher Aktivitäten, (2) dynamische Löschungen vorhandener Aktivitäten und (3) dynamische Rücksprünge zu bereits durchgeführten Aktivitäten, wie in Unterabschnitt 1.3.2 beispielhaft erläutert. Der Ansatz nutzt dabei die Flexibilität der Prozessdefinitions-Modellierungssprache WS-BPEL aus, um dynamische Änderungen nachzubilden. Vorhandene Prozessdefinitionsmodelle werden um zusätzliche Aktivitäten, Verzweigungen und Schleifen angereichert, deren Verhalten zur Laufzeit so gesteuert wird, dass aus Sicht eines Prozessbeteiligten die besagten dynamischen Änderungen möglich sind, obwohl die tatsächlich durch die Erweiterung des WebSphere Process Server nur simuliert werden.

Beiträge Die in dieser Arbeit beschriebene Erweiterung des bestehenden, statischen Prozessmanagementsystems WebSphere Process Server ist neuartig. Sie unterscheidet sich von verwandten Arbeiten und Vorgängerarbeiten dadurch, dass die Kontrolle auf allen Granularitätsebenen der unterstützen Prozesse gänzlich auf dem WebSphere Process Server verbleibt. Der WebSphere Process Server wird nur soweit ergänzt, dass dynamische Änderungen zur Laufzeit simuliert werden können. In Vorgängerarbeiten [Jäg03, HJK$^+$08] wurde das prototypisch entwickelte AHEAD-System stets nur um Funktionalitäten anderer, kommerzieller Systeme durch Kopplung erweitert bzw. bestimmte Funktionalitäten von AHEAD an externe Systeme ausgelagert. AHEAD bildet in diesen Arbeiten jedoch immer den zentralen Baustein des gesamten Prozessmanagementsystems.

1.4.4 Verwendung expliziten Prozesswissens

Der in Kapitel 3 beschriebene Simulationsansatz ermöglicht dynamische Änderungen in Prozessinstanzmodellen zur Prozesslaufzeit. Hierdurch entstehen jedoch neue Probleme: Insbesondere dynamische Änderungen können nicht beliebig erfolgen, sondern müssen technischen und fachlichen Rahmenbedingungen unterliegen wie in Unterabschnitt 1.3.3 dargelegt.

In Kapitel 4 wird ein Ansatz beschrieben, mittels dessen technische und fachliche Bedingungen in (dynamisch geänderten) Prozessmodellen werkzeuggestützt innerhalb eines Prozessmodelleditors überprüft werden können. Technische Rahmenbedingungen werden getestet, in dem die innere *Korrektheit* eines Prozessmodells überprüft wird, d.h. ein Prozessmodell wird gegen eine Reihe von festen, formalisierten Regeln geprüft. Korrektheitsprüfungen stehen dabei für Prozessmodelle auf allen drei Abstraktionsebenen zur Verfügung. Die Verletzung oder Einhaltung fachlicher Rahmenbedingungen wird festgestellt, indem Prozessinstanz- oder -definitionsmodelle auf *Komplianz* gegen ein Prozesswissensmodell geprüft werden. In Komplianzprüfungen wird also immer ein Paar bestehend aus einem Prozesswissensmodell einerseits und einem Prozessdefinitions- oder -instanzmodell andererseits geprüft. Sowohl Korrektheits- als auch Komplianzprüfungen führen zu detaillierten Fehlermeldungen im Prozessmodelleditor, die Prozessbeteiligte auf Art und Ort des Fehlers im jeweiligen Prozessmodell hinweisen.

Beiträge Die Verwendung expliziten Prozesswissens ist die konsequente Weiterführung von Vorgängerarbeiten. Krapp verzichtet in [Kra98] gänzlich auf die Berücksichtigung von bestehenden Prozessmodellierungsstandards, Schleicher verwendet in [Sch02] den Modellierungsstandard UML, davon jedoch Modellierungssprachen, in denen grundsätzlich Strukturen und nicht Prozesse, also Verhalten, modelliert werden. Die Verwendung des Standards WS-BPEL für Prozessdefinitionsmodelle beeinflusst den Entwurf der Sprache für Prozesswissensmodelle stark, nicht zuletzt aufgrund der höheren Flexibilität (Verzweigungen und Schleifen) von WS-BPEL.

Prozesswissensmodelle in dieser Arbeit unterscheiden sich in ihrer Semantik stark von vergleichbaren Modellen der Vorgängerarbeiten: Prozesswissensmodelle aus Vorgängerarbeiten (dort “Prozessdefinitionen auf Typebene” bzw. Aufgabenschemata genannt) treffen Aussagen über die mögliche syntaktische Struktur von Prozessinstanzmodellen (dort “DYNAMITE-Aufgabennetze” genannt). Hierüber kann beispielsweise ausgedrückt werden, dass in einem Prozessinstanzmodell eines Entwicklungsprozesses Entwurfsaktivitäten mit Implementierungsaktivitäten direkt über Kontrollflusskanten verbunden sein dürfen. Prozesswissensmodelle dieser Arbeit treffen Aussagen über das Verhalten, d.h. das erzeugte Transitionssystem eines Prozessdefinitions- oder -instanzmodells (vgl. Abschnitt 2.6). Die Semantik von Prozesswissensmodellen dieser Arbeit ist daher mit Bezug auf das in Prozessdefinitions- und -instanzmodellen modellierte Verhalten erklärt. In Prozesswissensmodellen dieser Arbeit kann beispielsweise ausgedrückt werden (vgl. Abbildung 1.5), dass nach einer Aktivität Gutachten beauftragen anschließend irgendwann, also nicht zwangsweise direkt im Anschluss, eine Aktivität Rücklauf einpflegen durchgeführt werden muss. Dabei ist unerheblich, ob im geprüften Prozessdefinitions- oder -instanzmodell beide Aktivitäten durch eine Kontrollflusskante direkt verbunden sind oder ihre Ausführungspräzedenz sich indirekt aus bestimmten Kombination verschiedener Kontrollflusskonstrukte ergibt. Trotzdem werden auch in dieser Arbeit werkzeuggestützte Prüfungen von Korrektheitsbedingungen als auch Komplianzbedingungen einheitlich anhand der Syntax der Prozessmodelle durchgeführt. Die zahlreichen Gründe hierfür werden in Abschnitt 4.7 erläutert.

1.4.5 Verwendung impliziten Prozesswissens

Die Modellierung von explizitem Prozesswissen in Prozesswissensmodellen verursacht zusätzlichen Aufwand. Dementsprechend besteht die Gefahr, dass Prozesswissensmodelle unvollständig sind und nicht gewartet werden, und Komplianzprüfungen gegen diese Modelle dementsprechend ebenso unvollständige oder fehlerhafte Ergebnisse liefern. Umgekehrt ist Prozesswissen in der – normalerweise großen – Menge vorhandener Prozessdefinitions- und -instanzmodelle implizit vorhanden. In Kapitel 5 wird beschrieben, wie dieses implizite Wissen durch Werkzeuge ausgenutzt werden kann. So genannte *Konsistenzprüfungen* decken Unterschiede insbesondere zwischen Prozessinstanzmodellen auf. Prozessbeteiligte erhalten hierüber Rückmeldung, ob eine dynamische Änderung im Widerspruch zu anderen, ähnlichen Prozessmodellen steht.

Beiträge Die Einbeziehung impliziten Prozesswissens ist gegenüber Vorgängerarbeiten gänzlich neuartig. Sie unterscheidet sich von Inferenzansätzen in [Sch02] insoweit, dass nicht die Gewinnung von Prozesswissensmodellen aus Prozessinstanzmodellen verfolgt wird, sondern der direkte Vergleich verschiedener Prozessinstanzmodelle. Der Vergleichsansatz abstrahiert von der Prozessmodellsyntax, d.h. basiert insbesondere nicht wie in vielen verwandten Arbeiten auf Grapheditierabständen zwischen Prozessinstanzmodellen. Stattdessen fußt er auf der Ausführungssemantik der Prozessinstanzmodelle, d.h. wertet aus Prozessinstanzmodellpaaren erzeugte Transitionssysteme aus. Insbesondere die Berücksichtigung der bisherigen Prozesshistorie in Prozessinstanzmodellen beim Vergleich sowie die Verwendung von Graphgrammatiken zur Erzeugung eines Transitionssystem, welches Basis einer detaillierten Vergleichsanalyse ist, unterscheiden den Ansatz dieser Arbeit von sämtlichen verwandten Arbeiten.

Kapitel 2

Eine Prozessmodellschichtung

In Kapitel 1 wurden Prozessmodelle auf unterschiedlichen Abstraktionsniveaus verwendet: Prozessinstanzmodelle modellieren konkrete, in Ausführung befindliche Prozesse, Prozessdefinitionsmodelle Prozesstypen und Prozesswissensmodelle domänenspezifisches Prozesswissen, das prozesstypübergreifend ist. In diesem Kapitel werden Sprachen für diese verschiedenen Prozessmodellarten auf formalem Wege definiert. Dabei wird der Sprachstandard Web Services Business Process Execution Language (WS-BPEL) [AAA^{+}07] als Ausgangspunkt der Sprachdefinitionen verwendet.

Dieses Kapitel ist wie folgt gegliedert: In Abschnitt 2.1 werden grundlegende Begriffe und Aspekte der Prozessmodellierung eingeführt sowie Verfahrensweisen zur Sprachdefinition erörtert, auf denen die anschließenden Abschnitte aufsetzen. Abschnitt 2.2 führt die in dieser Arbeit entwickelten Prozessmodellierungssprachen beispielhaft und auf informale Weise ein. In Abschnitt 2.3 werden die Syntaxen der unterschiedlichen Prozessmodellierungssprachen formal definiert und dabei syntaktische Querbezüge und Überlappungen der Sprachen präzise dargelegt. Die Abschnitte 2.4 und 2.5 erläutern die formalen Semantiken der Sprachen.

Die nachfolgenden Kapitel bauen auf den Erläuterungen in diesem Kapitel auf: In Kapitel 3 werden Prozessinstanzmodelle als Kommunikationsmittel zwischen Prozessmanagementsystem und Prozessbeteiligten zur Kommunikation von Prozesszuständen und dynamischen Prozessänderungen verwendet. Kapitel 4 beschreibt, wie Inkorrektheiten innerhalb von Prozessmodellen und Inkomplianzen zwischen verschiedenen Prozessmodellen durch syntaktische Analysen aufgefunden werden können. In Kapitel 5 wird gezeigt, wie die formale Semantik von Prozessinstanzmodellen ausgenutzt werden kann, um Gemeinsamkeiten und Unterschiede zwischen Prozessinstanzmodellen aufzudecken. Das Verständnis der Prozessmodellsyntax und -semantik ist also erforderlich, um die Erläuterungen in den nachfolgenden Kapiteln vollständig nachvollziehen zu können.

2.1 Prozessmodellgrundlagen

Modellbildung ist in der Informatik und insbesondere im Bereich des Prozessmanagements ein gängiger und häufig notwendiger Vorgang. Auch diese Arbeit befasst sich mit verschiedenartigen Prozessmodellen. In diesem Abschnitt werden daher grundlegende Eigenschaften und Begriffe im Umfeld von Modellen und der Modellierung erläutert.

2.1.1 Merkmale von Modellen

Modelle zeichnen sich nach Stachowiak [Sta73] durch folgende Merkmale aus:

Abbildungsmerkmal Modelle sind stets *Abbildungen* von *Originalen*. Die Originale können verschiedenartiger *Natur* sein, beispielsweise materieller Natur wie Bauten oder Maschinen oder immaterieller Natur wie Software. Die Originale im Prozessmanagementbereich sind Arbeitsabläufe.

Verkürzungsmerkmal Ein Modell unterscheidet sich absichtlich von seinem Original. Typischerweise fehlen einem Modell eine Vielzahl von Eigenschaften (*Attribute*) des Originals, die für den jeweiligen Zweck des Modells irrelevant sind. In Prozessmodellen beispielsweise wird zwar festgehalten, welche Ressource eine bestimmte Aktivität durchführt, aber beispielsweise nicht, wo diese Aktivität durchzuführen ist, da diese Angabe für die Prozessdurchführung i.A. irrelevant ist. Im zugehörigen Original ist der Ort jedoch ein feststellbares Attribut. Die nur im Original vorhandenen Attribute nennt man auch *präterierte* Attribute. Umgekehrt gibt es mitunter auch so genannte *abundante* Attribute, die nur im Modell vorkommen, nicht jedoch im Original.

Pragmatisches Merkmal Zu einem Original gibt es nicht ein einziges denkbares Modell. Vielmehr richtet sich die Modellbildung nach einem bestimmten *Modellzweck*, beispielsweise Dokumentation, Ausführungsunterstützung, Simulation oder Analyse eines Originals. Bei komplexen Originalen und insbesondere bei solchen, die erst entwickelt werden sollen, ist es gängige Praxis, das Original schrittweise und getrennt in verschiedenen Aspekten zu modellieren. Idealisiert betrachtet transformieren Softwareentwicklungsprozesse beispielsweise abstrakte (detailarme) und informale Anforderungen durch fortwährende Detaillierung und Formalisierung in ausführbare Programme. Hierbei werden verschiedene Aspekte wie die Modulstruktur oder das Laufzeitverhalten zumindest als Zwischenprodukte in unterschiedlichen Modellen festgehalten. Gleiches gilt für Prozessmanagementsysteme. Abläufe werden zumeist auf rein fachlicher

Ebene festgehalten, idealerweise soweit, bis sich alle fachlichen Details im Modell wiederfinden. Erst dann wird das Modell formalisiert, häufig durch händische oder semiautomatische Übersetzung von einer nicht- oder semiformalen Modellierungssprache in eine formale.

Nach Stachowiak bilden also Modelle Originale verkürzt zu einem bestimmten Zweck ab. Die Charakterisierung von Modellen allein anhand der Verkürzung und des Modellzwecks ist jedoch zu grob. Vielmehr weisen Modelle und insbesondere Prozessmodelle Charakteristika auf, die sich wie folgt klassifizieren lassen.

Aspekt Modelle werden im Allgemeinen nicht wahllos gegenüber dem Original verkürzt. Stattdessen ist die Verkürzung häufig derart, dass Modelle nur bestimmte Aspekte eines Originals modellieren und andere auslassen. In Unterabschnitt 2.1.3 wird hierauf in Hinblick auf Prozessmodelle näher eingegangen.

Abstraktheit Die Abstraktheit ist ein Maß dafür, inwieweit Originaldetails im Modell ausgelassen wurden. Bei Prozessmodellen wird hierfür auch synonym der Begriff *Granularität* verwendet.

Formalheit Die Formalheit eines Modells wird dadurch bestimmt, inwieweit die Modellierungssprache, in der das Modell ausgedrückt ist, formal definiert ist. Unterabschnitt 2.1.2 geht hierauf näher ein.

Sprachebene Spezialfälle stellen Modelle von Sprachen dar, in denen wiederum Modelle ausgedrückt werden können. Diese werden einer höheren Sprachebene zugeordnet als die erstgenannten und werden in der Literatur häufig als Metamodelle bezeichnet. Unterabschnitt 2.1.4 vertieft dieses Charakteristikum.

Versicherungsprozesse stellen wie Prozesse allgemein *immaterielle Originale* dar. Die modellierten *Aspekte* eines Prozesses sind vornehmlich der Informations- und Verhaltensbezogene Aspekt, der in Aktivitätsmodellen festgehalten wird, der Organisatorische Aspekt (Ressourcenmodelle) und der Produktbezogene (Produktbezogene). Hierauf wird in Unterabschnitt 2.1.3 näher eingegangen. Versicherungsprozesse können unterschiedlich abstrakt modelliert werden. Da gerade bei diesen Prozessen im Gegensatz zu beispielsweise Entwicklungsprozessen die Automatisierung ein vorrangiges und machbares Ziel ist, sind die hier eingesetzten Modellierungssprachen *formal*, beispielsweise die in Unterabschnitt 2.1.5 beschriebene WS-BPEL.

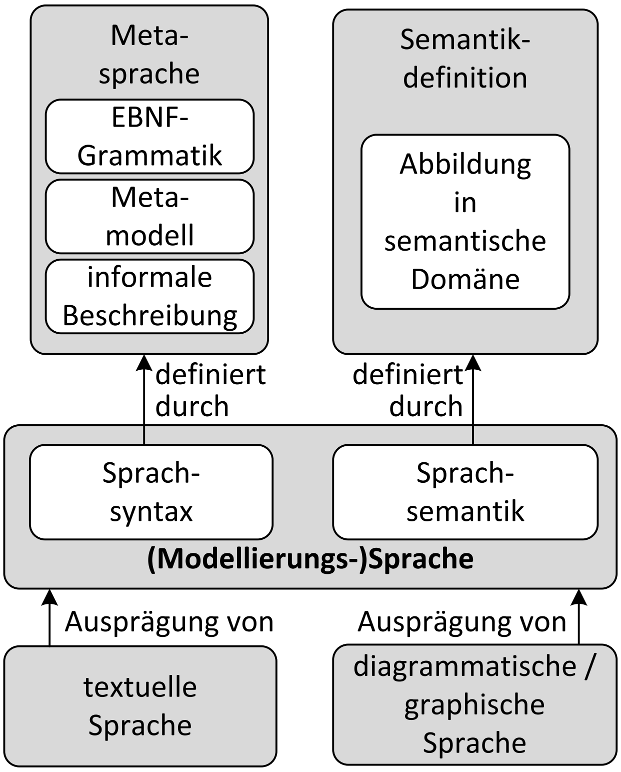


Abbildung 2.1: Beziehungen zwischen Sprache, Syntax und Semantik

2.1.2 Semiotik von Modellierungssprachen

Modelle dienen dazu, Informationen über ein Original festzuhalten und zu kommunizieren. Voraussetzung hierfür ist, dass es eine *Vereinbarung* über den *Aufbau (Syntax)* und die *Bedeutung (Semantik)* von Modellen gibt. Es muss also vereinbart werden, welche Arten von Modellelementen in welcher Kombination überhaupt verwendet werden können, wie diese notiert werden und was ihre Bedeutung in Bezug auf ein Original ist. Eine Menge solcher Vereinbarungen wird als *Modellierungssprache* bezeichnet (vgl. Abbildung 2.1).

Bei bestimmten Modellen, die nur von Mensch zu Mensch kommuniziert werden, beispielsweise bei zweidimensionalen Projektionen von materiellen Gegenständen, kann die Modellierungssprache *implizit* bleiben. Jeder menschliche Betrachter eines solchen Modells kann dank angeborener Fähigkeiten ein solches Modell interpretieren, d.h. die richtige gedankliche Vorstellung vom Original erzeugen. Bei immateriellen Originalen, beispielsweise Prozessen, ist normalerweise eine *explizite* Festlegung der Modellierungssprache notwendig.

Werden Modelle zur Kommunikation zwischen Mensch und Computer eingesetzt, d.h. werden die Modelle in irgendeiner Form verarbeitet, muss die Modellierungssprache eine *formale Sprache* sein. Das bedeutet, dass die Festlegungen der Modellierungssprache explizit und exakt sein müssen. Auf

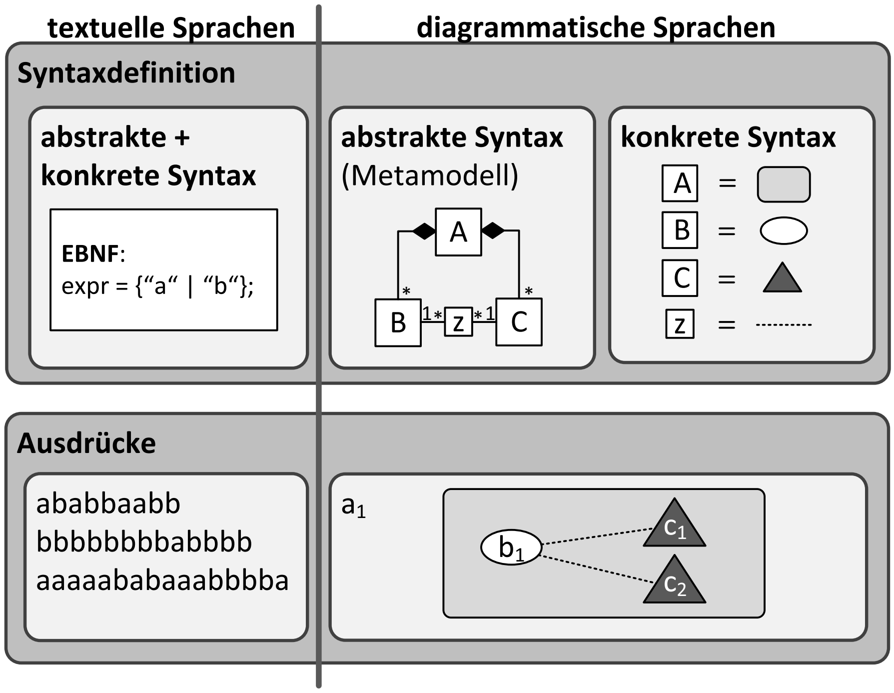


Abbildung 2.2: Textuelle und diagrammatische Sprachen

natürliche Sprachen trifft dies nicht zu.

Die Festlegungen betreffen zwei Aspekte einer Modellierungssprache: Die Syntax und Semantik. Die folgenden Unterabschnitte erläutern diese beiden Aspekte.

Syntax einer Modellierungssprache

Die Syntax einer Sprache beschreibt den Aufbau gültiger Ausdrücke in dieser Sprache. Sie kann selbst wiederum in einer Metasprache definiert werden. Je nachdem, ob die Syntax einer diagrammatischen Sprache (häufig auch visuelle Sprache genannt) oder einer textuellen Sprache definiert werden soll, kommen verschiedene Metasprachen zum Einsatz.

Textuelle Sprachen Ausdrücke (Wörter) textueller Sprachen sind Sequenzen von Zeichen (Symbolen) aus einer bestimmten Grundmenge, dem Alphabet [MS97, Abschnitt 2.1]. Zur Syntaxdefinition von textuellen Sprachen werden normalerweise formale textuelle Metasprachen und natürliche Sprachen verwendet. Der Standard für Erstere ist die Extended Backus Naur Form [Int96], mit der kontextfreie Grammatiken formal ausgedrückt werden können, also Grammatiken, die die lexikalische und kontextfreie Syntax-Ebenen [HMU02, Kapitel 9] einer textuellen Sprache formal beschreiben. Für viele textuelle Sprachen sind die Beschreibung kontextfreier Grammatiken zu ausdrucksschwach; daher werde sie im Allgemeinen um weitere Regeln in natürlicher Sprache erweitert. Dies trifft beispielsweise auf Programmiersprachen wie Java [GJS05, Kapitel 18] oder C++ [Int03, Seite 675ff] zu.

Beispiel 2.1 (Syntax einer textuellen Sprache) Abbildung 2.2 zeigt auf der linken Seite ein einfaches Beispiel für eine Syntaxdefinition in EBNF. Die definierte Sprache sind hierbei Zeichenketten, die nur aus den Symbolen a und b bestehen.

Diagrammatische Sprachen Im Bereich des Prozessmanagements sind diagrammatische Sprachen für die Modellierung üblich. Während für einen textuellen Ausdruck die Vorkommnisse und Reihenfolge der Zeichen signifikant sind, wird ein diagrammatischer Ausdruck durch die Vorkommnisse geometrischer Objekte und deren Anordnung (üblicherweise im zweidimensionalen Raum) festgelegt. Im Gegensatz zu textuellen Sprachen werden bei der Syntaxdefinition diagrammatischer Sprachen üblicherweise abstrakte und konkrete Syntax getrennt beschrieben. Die Definition der *abstrakten Syntax* definiert, welche Symbole in den Sprachausdrücken (hier: Diagramme) vorkommen und welche Symbole mit welchen anderen in Beziehung stehen. Die Definition der *konkrete Syntax* beschreibt einerseits, wie die Symbole als geometrische Objekte dargestellt werden sollen und andererseits wie Beziehungen zwischen Symbolen durch Anordnungen der geometrischen Objekte repräsentiert werden sollen.

Beispiel 2.2 (Syntax einer diagrammatischen Sprache)
Auf der rechten Seite von Abbildung 2.2 ist die Syntax einer einfachen rein diagrammtischen Sprache angegeben. Sie besteht laut der Definition der abstrakten Syntax aus den Symbolen A, B, C und z, wobei die Paare (A,B), (A,C), (A,z) sowie (z,C) jeweils in Beziehung zueinander stehen. Die Definition der konkreten Syntax gibt an, dass beispielsweise ein Symbol A als abgerundetes Rechteck dargestellt werden soll, und das Symbol z als strichlierte Linie. Darüber hinaus gibt diese Definition an, dass eine Beziehung zwischen einem A-Symbol und einem B-Symbol so dargestellt wird, dass das A-Rechteck die betreffende B-Ellipse visuell enthält. In dem Beispielausdruck enthält a_1 alle übrigen Elemente. Beziehungen können auch durch visuelle Inzidenz dargestellt werden, beispielsweise zwischen z-Linien und B-Ellipsen.

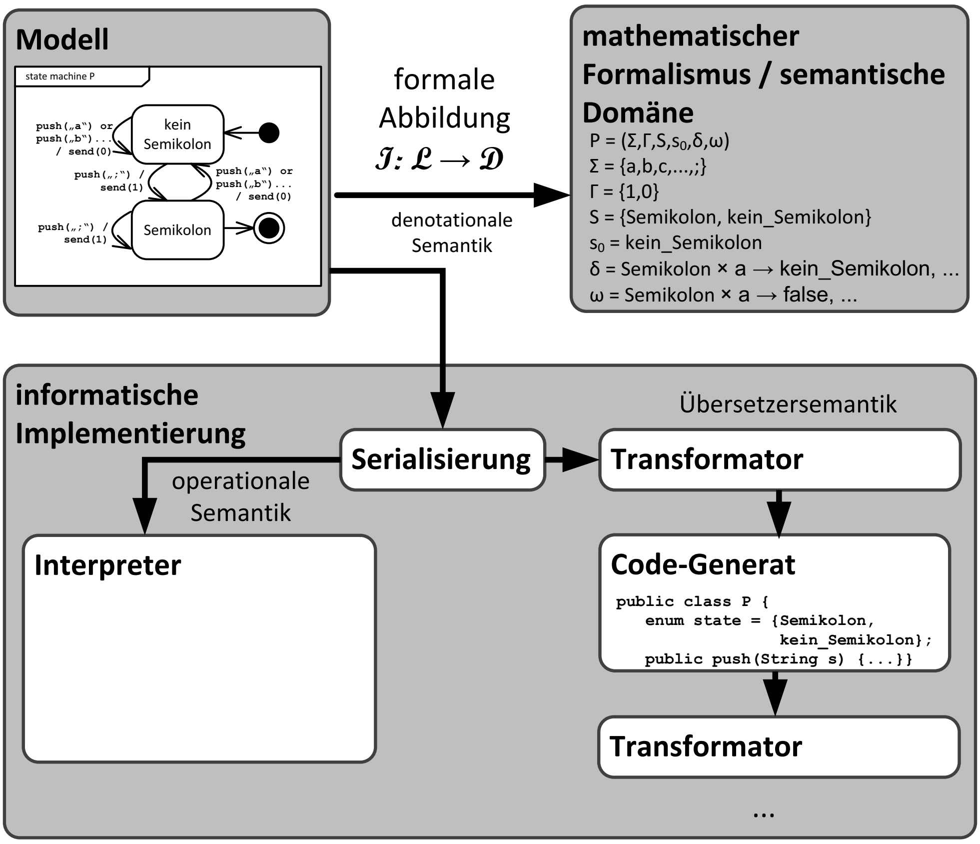


Abbildung 2.3: Ansätze zur Semantikdefinition

Das Beispiel 2.2 für die diagrammatischen Sprachen ist insoweit vereinfacht, als dass zumindest Prozessmodelle nie Ausdrücke rein diagrammatischer Sprachen sind, sondern Ausdrücke aus textuell-diagrammatischen Mischformen mit dominierendem diagrammatischen Anteil. Textuelle Anteile in den Modellen finden sich beispielsweise in den Benennungen der Symbole, im Modellkontext auch *Modellelemente* genannt, wieder oder als Bedingungen an Kontrollflussdefinitionen. Diese textuellen Anteile unterliegen normalerweise auch einer (einfachen) Syntax.

Semantik einer Modellierungssprache

Neben der Syntax einer Modellierungssprache, also dem Aufbau gültiger Ausdrücke (Modelle) der Sprache, ist die Bedeutung der Modelle zu klären. Sollen Modelle, beispielsweise die auf “state charts” [Har87] basierenden UML-Zustandsdiagramme, in irgendeiner Form maschinell verarbeitet werden, muss die Semantik formal definiert werden.

Zur Definition der Semantik einer bestimmten formalen Sprache M gibt es verschiedene Ansätze [NN92], die in Abbildung 2.3 skizziert sind. Je nach Ansatz spricht man dann von einer bestimmten Semantik von M. Häufig wird eine Sprache M über eine *Übersetzersemantik* [SK95, Kapitel 7] erklärt, indem eine Abbildung der Sprachkonstrukte von M auf eine "alte" Sprache "O" definiert wird, deren Semantik bereits geklärt ist. Eine direkte Anwendung einer solchen Semantik findet sich im Compilerbau (Übersetzerbau). Ein UML-Zustandsdiagramm (in serialisierter Form) kann beispielsweise durch einen Transformator in Java-Code überführt werden. Ein anderer Ansatz ist die *operationale Semantik* [SK95, Kapitel 8], die auf eine abstrakte Maschine Bezug nimmt. Diese operationale Semantikdefinition kann dann als Vorlage für einen Interpreter dienen. In einer *denotationalen Semantikdefinition* [SK95, Kapitel 9] werden die Konstrukte von M durch eine formale Abbildung I in einen mathematischen Formalismus abgebildet.

Für viele gängige formale Sprachen gibt es formale Semantikdefinitionen. Die Semantik der Datenbankabfragesprache SQL beispielsweise kann über die Relationale Algebra definiert werden [CG85] oder über prädikatenlogische Kalküle [NPS91]. Die Programmiersprache Java lässt sich über eine denotationale Semantik [AFL99] erklären. Für die Sprache PROGRES, in der programmierte Graphersetzungssysteme spezifiziert werden können, existiert eine Semantikdefinition [Sch91], die die syntaktischen Spezifikationen in den mathematischen Formalismus der programmierten Graphersetzungssysteme [Nag79] abbildet.

Auch für die Vielzahl existierender Prozessmodellierungssprachen – WS-BPEL, EPKs und BPMN gehören derzeit zu den gängigsten – gibt es Arbeiten, die eine formale Definition der betreffenden Semantik zum Ziel haben. Die meisten bilden dabei die jeweiligen Sprachkonstrukte auf Petri-Netze [Rei00] ab. In [OVA$^+$07] geschieht dies beispielsweise für WS-BPEL (vgl. Unterabschnitt 2.1.5). EPKs können zu YAWL-Modellen [AH05] transformiert werden [MMN06], die selbst wiederum über Petri-Netze erklärt sind. Auch die Semantik von BPMN-Modellen kann über eine Abbildung in Petri-Netze definiert werden [DDO07]. Alternativ können BPMN-Modelle auch in Communicating Sequential Processes (CSP) [Hoa85] übersetzt werden [WG08], wobei die diagrammatische Syntax von BPMN-Modellen zunächst in die textuelle Z-Notation [Spi89] übersetzt wird.

Präzision eines Prozessmodells

Verschiedene Prozessmodelle können stark in ihrer *Präzision* variieren. Die Präzision eines Modells kann hierbei aus zwei Blickwinkeln betrachtet werden, d.h. der Begriff der Präzision zerfällt genau betrachtet in zwei Unterbegriffe [KER99, Abschnitt Q2.2.]): Die *Abstraktheit* bzw. im Gegenteil die *Detailliertheit* eines Modells bezieht sich auf die Präzision der Aussagen des Modells selbst, die *Formalheit* bzw. *Informalheit* auf die Präzision der verwendeten

Sprache.

Kent et al. führen als Beispiel formale mathematische Ausdrücke von Intervallen auf, die insoweit abstrakt sind, als dass sie einen eigentlich gemeinten Wert nur eingrenzen [KER99]. Für diagrammatische Modelle gilt Ähnliches: Beispielsweise ist es möglich, die Semantik bestimmter UML-Modellarten formal festzulegen [Rum04b]. Diese Formalisierung ist Voraussetzung für die Übersetzung von UML-Modellen in eine Programmiersprache wie Java [Rum04a, Abschnitt 4]. Aus UML-Modellen generierte Java-Klassen können nach der Übersetzung weiter detailliert werden. In [RT98] zeigen Rumpe et al., dass eine Formalisierung einer diagrammatischen Sprache auch dann nötig ist, wenn Modelle in dieser Sprache fortwährend detailliert werden. Die Autoren entwickeln hierfür ein Kalkül, über dessen Regeln datenflussorientierte Prozessmodelle so verfeinert werden können, dass bestimmte Korrektheitsaspekte erhalten bleiben.

2.1.3 Teilmodelle und Prozessaspekte

Das Management von Prozessen ist schwierig, da sie kompliziert und komplex im Sinne von [RR04, Kapitel 1] sind. *Komplex* sind Prozesse deswegen, weil die Prozessbestandteile auf vielschichtige Art miteinander wechselwirken können. Auch augenscheinlich einfache Prozesse können insoweit komplex sein, als dass sie schwer vorhersehbares Verhalten aufweisen können [CM84]. Die Komplexität von Prozessen kann nur durch rigide Analyse beherrscht werden, die aber durch die Anwendung mathematischer Formalismen (s. Abschnitt 2.4) erleichtert wird.

Kompliziert, also umfangreich und daher schwer zu überblicken, sind Prozesse deswegen, weil viele Aktivitäten ausgeführt werden müssen, dabei viele (Zwischen-)Produkte (z.B. Daten) entstehen und viele Ressourcen daran beteiligt sind. Der Kompliziertheit von Prozessen kann man durch gängige Prinzipien wie Modularisierung oder Abstraktion begegnen. Des Weiteren ist es hilfreich, verschiedene Prozessaspekte trotz ihrer wechselseitigen Abhängigkeiten voneinander getrennt zu betrachten. Für die Modellierung von Prozessen bedeutet das, dass Prozessmodelle in Teilmodelle gegliedert werden, die jeweils einen bestimmten Prozessaspekt fokussieren.

Dieser Unterabschnitt beschreibt eine mögliche Aufteilung von Prozessmodellen in Teilmodelle. Diese Aufteilung ist [NW94] entnommen. Auf sie wird in den nachfolgenden Kapiteln Bezug genommen.

Die Abbildung 2.4 fasst die wesentlichen Teilmodelle zusammen. Sie zeigt dabei exemplarisch einen Ausschnitt eines Entwicklungsprozesses einer verfahrenstechnischen Anlage. Die Unterteilung ist angelehnt an die aus [HJK+08, Abb. 3.61.]. Die Teilmodelle korrespondieren dabei mit der im Prozessmanage-

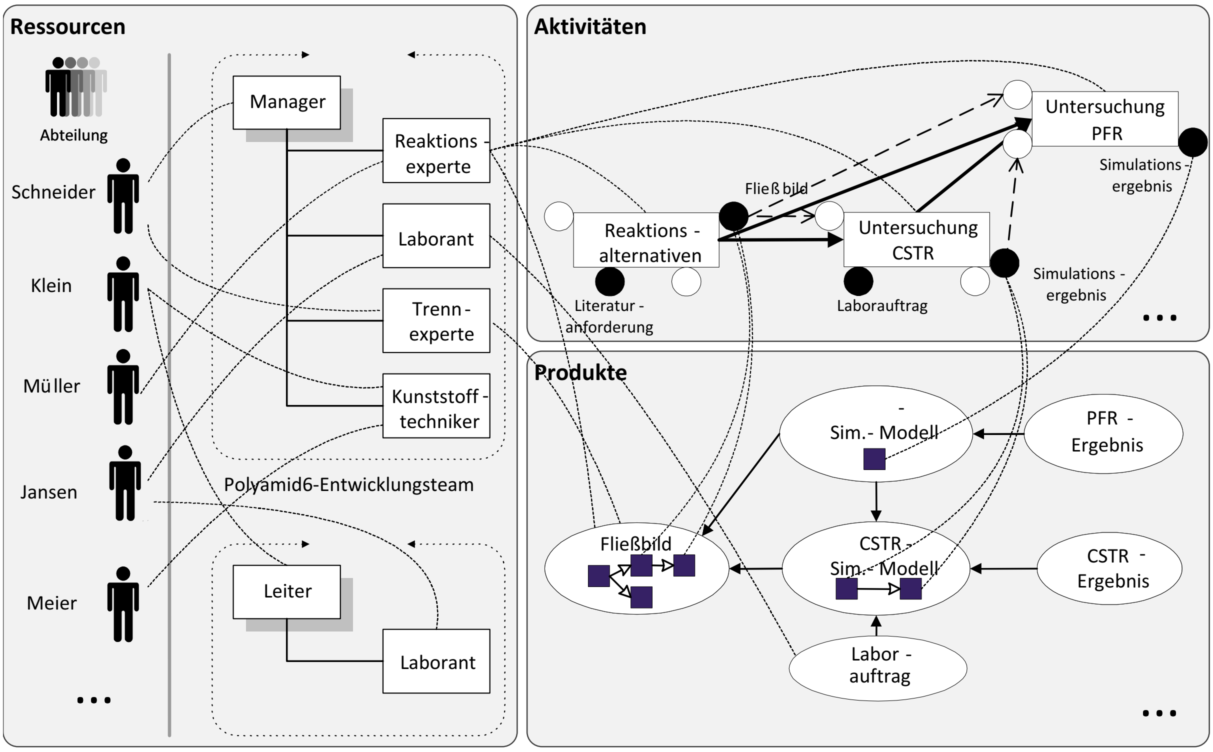


Abbildung 2.4: Prozess-Teilmodelle (nach [HJK+08, Abb. 3.61.])

ment gängigen Aufteilung in *Prozessaspekte* (vgl. Abschnitt 1.1, [JB96, Kapitel 6] sowie [Joe00, Unterabschnitt 2.1.3]) und werden im Folgenden erläutert. Prozessaspekte sind hierbei nicht mit *Sichten* zu verwechseln. Sichten sind spezielle Modelle, die beispielsweise bei der Modellierung von Softwaresystemen erstellt werden. Typisch für Sichten ist aber, dass Modellelemente verschiedener Sichten sich auf das gleiche Teil des Originals beziehen, wobei unterschiedliche Details betont werden, also unterschiedliche Attribute präteriert oder abundant sind. Modellelemente in unterschiedlichen Teilmodellen hingegen beziehen sich auch auf unterschiedliche Originalteile.

Aktivitäten

Ein zentraler Aspekt bei der Modellierung von Prozessen ist der *Funktionale Aspekt* und der *Verhaltensaspekt*. Ersterer findet sich im Bereich Aktivitäten der Abbildung insoweit wieder, als dass hier modelliert wird, dass ein verfahrenstechnischer Entwicklungsprozess (unter anderem) aus Aktivitäten[1] zur Bestimmung von Reaktionsalternativen, zur Untersuchung eines CSTR (continuously stirring reactor) und eines PFR (plug flow reactor) besteht.

Neben der rein funktionalen Dekomposition des Entwicklungsprozesses

[1] Aktivitäten werden im Bereich der Entwicklungsprozesse häufig auch synonym mit *Aufgaben* bezeichnet.

in Aktivitäten und Unteraktivitäten wird in dem Teilmodell Aktivitäten auch das *Verhalten* des Prozesses modelliert. Dies geschieht über *Kontrollflüsse*, die im Wesentlichen festlegen, in welcher Reihenfolge die Aktivitäten ausgeführt werden. Kontrollflüsse bedeuten also *zeitliche Abhängigkeiten* zwischen Aktivitäten. Im konkreten Fall der Abbildung ist durch einen so genannten simultanen Kontrollfluss modelliert, dass die Aktivität Untersuchung CSTR erst nach der Beendigung von Reaktionsalternativen beendet werden darf und die Untersuchung PFR erst nach Reaktionsalternativen und Untersuchung CSTR.

Datenflüsse modellieren im Wesentlichen *produktbezogene Abhängigkeiten* zwischen Aktivitäten. Ein Datenfluss von einer Aktivität A zu einer Aktivität B wird genau dann modelliert, wenn bei der Durchführung von Aktivität A Produkte produziert werden, die für die Durchführung von Aktivität B notwendig sind. In Abbildung 2.4 verläuft beispielsweise ein Datenfluss von Reaktionsalternativen zu Untersuchung PFR; dieser steht für die auf das Produkt Fließbild bezogene Abhängigkeit der beiden Aktivitäten.

Ressourcen

Im Teilmodell Ressourcen steht der *Organisatorische Aspekt* im Fokus. Hier ist festgehalten, in welcher organisatorischen Beziehung *konkrete Ressourcen* untereinander stehen. Konkrete Ressourcen stehen dabei immer für tatsächliche Mitarbeiter oder Hard- und Software-Systeme, z.B. Entwicklungswerkzeuge. Beziehen sich Ressourcen auf Systeme, so werden diese Modellteile in der Literatur auch dem *Werkzeugbezogenen Aspekt* zugeordnet.

Planressourcen abstrahieren von konkreten Ressourcen. Aktivitäten können Planressourcen zugeordnet werden. Diese Zuordnung besagt, welche Befähigungen für die Ausführung einer Aktivität vorausgesetzt werden. Des Weiteren können konkrete Ressourcen Planressourcen zugeordnet werden, womit Berechtigungen und Befähigungen konkreter Ressourcen ausgedrückt werden.

Das Ressourcenmodell aus Abbildung 2.4 ist vereinfacht dargestellt und zeigt daher nur die wichtigsten Elemente.

Produkte

Unter Produkten sind die Artefakte zu verstehen, die während der Ausführung der Aktivitäten in einem Prozess von den jeweiligen Ressourcen produziert werden. Sie sind insoweit dem *Informationsaspekt* eines Prozesses zuzuordnen. In Entwicklungsprozessen handelt es sich bei Produkten zumeist um komplexe Dokumente, beispielsweise Stücklisten oder Fließbilder. In Geschäftsprozessen sind die Produkte mehr oder minder komplexe Daten, beispielsweise ein zweiwertiges Datum für eine Entscheidung oder ein Datenverbund, der eine Schadensakte repräsentiert.

Produkte stehen untereinander in vielfältiger Beziehung [Wes99b, Kapitel 2]. In Entwicklungsprozessen wird üblicherweise ein Produkt mehrfach überarbeitet, wodurch *Revisionen* erzeugt werden, oder ausgehend von einem bestimmten Entwicklungsstand in zwei *Varianten* mit unterschiedlichem Zweck aufgespalten. Diese Sachverhalte werden durch Versionsbeziehungen zwischen Produktversionen modelliert.

Zwischen unterschiedlichen Produkten können außerdem inhaltliche Abhängigkeiten bestehen. Diese werden als Beziehungen zwischen verschiedenen Produkten modelliert. Diese Beziehungen können als Hinweise für Integrationswerkzeuge [Bec07, Wör04] verwendet werden, die zur Wahrung der Konsistenz zwischen abhängigen Dokumenten verwendet werden.

Produktabhängigkeiten zwischen Aktivitäten induzieren teilmodellübergreifende Abhängigkeiten zwischen Aktivitäten und Produkten. Einer Aktivität kann dabei eine Menge von Produktversionen zugeordnet werden, die die betreffende Aktivität entweder produziert oder von der sie als Eingabe abhängt.

2.1.4 Modellierungssprachebenen

In den vorhergehenden Unterabschnitten wurden die im Bereich des Prozessmanagements gängige Semiotik und Prozessaspekte erläutert. Bereits in Unterabschnitt 2.1.2 wurden dabei unterschiedliche Modellebenen angesprochen, nämlich die zur Syntaxdefinition von diagrammatischen Modellen gebräuchlichen Metamodelle und die Modelle selbst. Gerade in diesem Fall ist die Trennung der Modellebenen etwas grundsätzlich anderes als die Trennung eines Prozessmodells in Teilmodelle, wie es in Unterabschnitt 2.1.3 erläutert wurde.

Das Vier-Ebenen-Metamodellierungsmodell

Metamodelle sind zur Syntaxdefinition von diagrammatischen Sprachen üblich, wie in Unterabschnitt 2.1.2 beschrieben. Hierüber ergeben sich zwei Ebenen der Metamodellierung: das Modell selbst und sein zugehöriges Metamodell. Nun ist das Metamodell selbst wiederum nichts weiter als ein Modell, das in einer bestimmten Sprache verfasst ist.

Diese Überlegung führt zunächst zu einer unendlichen Kette von Modellen und zugehörigen Metamodellen. Somit erscheint es zunächst unmöglich, die Syntax eines Modells vollständig zu klären, weil die Klärung einer Modellebene erzwingt, dass auch die jeweils darüber liegende Ebene erklärt werden muss. Diese unendliche Kette kann dadurch vermieden werden, dass eine Mo-

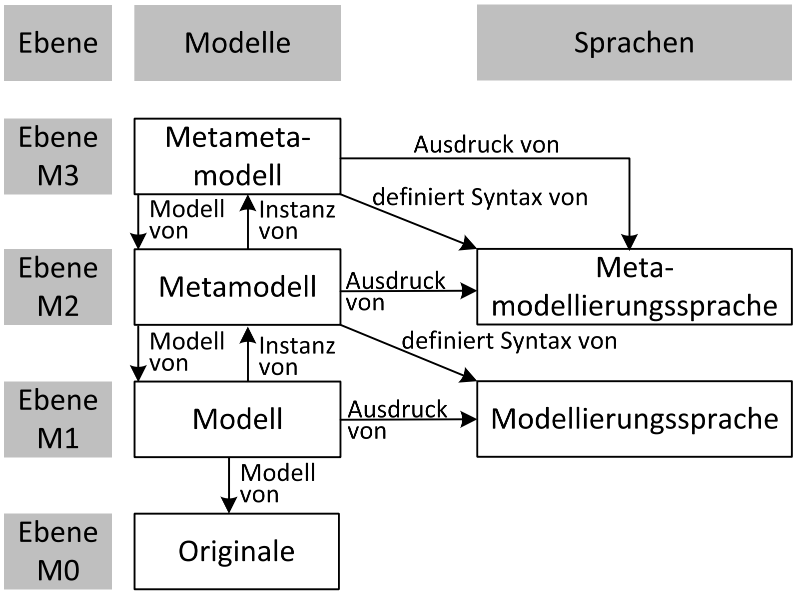


Abbildung 2.5: Vier-Ebenen-Metamodellierungsmodell (nach [KK02, Abb. 3])

dellierungssprache auf einer bestimmten Ebene durch sich selbst modelliert werden kann.

In der Literatur haben sich Schichtenmodelle mit vier Ebenen durchgesetzt [GKP98, KK02, Obj04]. Wie Abbildung 2.5 zeigt, sind die Ebenen mit M0 bis M3 durchnummeriert. Die unterste Ebene M0 ist die des letztendlich zu modellierenden Originals, beispielsweise ein Prozess, ein Prozesstyp oder Prozesswissen. Dieser Ebene wären beispielsweise der Prozesstyp und der Prozess aus der Anwendungswelt in Abbildung 1.2 zuzuordnen. Die Ebene M1 beinhaltet Modelle des Originals. Diese Modelle sind Ausdrücke einer Modellierungssprache. Die Syntax der Modellierungssprachen wird durch ein Metamodell auf Ebene M2 definiert. Das Metamodell selbst ist ein Ausdruck einer Metamodellierungssprache, deren Syntax wiederum in einem Metametamodell auf Ebene M3 dargelegt ist. Dieses Metametamodell kann derart gestaltet werden, dass es seine eigene Syntax definiert. Auf höhere Modellebenen kann also verzichtet werden. Wenn in der Literatur die Beziehung zwischen Modellen auf benachbarten Ebenen beschrieben wird, wird das Modell auf der höheren Ebene Mi verkürzt als Modell des Modells auf Ebene $M(i-1)$ bezeichnet. Diese Auffassung des Modellbegriffs weicht von der nach [Sta73] ab, wird aber im Folgenden in diesem Zusammenhang verwendet.

Modellierungssprachen der OMG

Die Modellierungssprachen der Object Management Group (OMG) sind an dem 4-Ebenen-Schichtenmodell ausgerichtet (s. Abbildung 2.6). Die Originale auf M0-Ebene sind hierbei beispielsweise Softwaresysteme. Softwaresysteme können in verschiedenen Modellen auf M1-Ebene modelliert werden. Klassendiagramme können beispielsweise beim Entwurf eines Softwaresystems verwendet werden. In ihnen wird die statische Modul- bzw. Klassenstruktur dieses Systems modelliert. Die Modelle der M1-Ebene sind beispielsweise in der Unified Modeling Language (UML) [GK01]) verfasst. Der Begriff “unified” (deutsch: vereinheitlicht) deutet dabei darauf hin, dass die UML eine gemeinsame Sprache für die verschiedenen Modellarten (Klassendiagramme, Objektdiagramme, Sequenzdiagramme etc.) auf M1-Ebene ist. Die Syntax der UML ist durch das UML-Metamodell auf Ebene M2 festgelegt. Das UML-Metamodell definiert beispielsweise den Aufbau einer Klasse in einem Klassendiagramm, zu dem insbesondere Attribute und Operationen der Klasse gehören. Das UML-Metamodell ist ein Ausdruck der Sprache Meta Object Facility (MOF) [Obj04], deren Syntax wiederum durch das MOF-Metamodell auf Ebene M3 definiert ist, das reflexiv also selbst ein MOF-Ausdruck ist.

Einerseits kann die MOF also verwendet werden, um Metamodelle auszudrücken. Deshalb ist das Metamodell der MOF auf Ebene M3 anzusiedeln und somit ein Meta-Metamodell. Andererseits können abseits des UML-Kontextes MOF-Ausdrücke auch direkt Modelle auf Ebene M2 sein. Deshalb befindet sich die MOF gleichsam auf Ebene M2 [Gro05, Abschn. 7.9]. Des Weiteren ist das UML-Metamodell nicht nur ein MOF-Ausdruck, MOF verwendet auch gewisse Bestandteile des UML-Metamodells, die unter dem Begriff Infrastructure Library zusammengefasst sind. Daher wäre es auch möglich, das UML-Metamodell reflexiv zu definieren. Im OMG-Standard wird jedoch aus Gründen der Einheitlichkeit der Umweg über die MOF gegangen, da in MOF auch andere Metamodelle formuliert sind, auf die die Reflexivität nicht zutrifft, beispielsweise das Common Warehouse Metamodel (CWM) [Gro03].

Das MOF-Metamodell ist selbst in zwei Bestandteile unterteilt. Die Teile innerhalb des *Essential MOF (EMOF)* definieren eine Subsprache von MOF, mittels derer vereinfachte Klassendiagramme ausgedrückt werden können. Das vollständige MOF-Metamodell wird genauer mit *Complete MOF (CMOF)* bezeichnet. Unter anderem kommen gegenüber EMOF Modellierungskonstrukte wie Assoziationen mit Navigierbarkeitsangaben hinzu.

Die Abbildungen 2.5 und 2.6 dienen der Verdeutlichung der Beziehung zwischen verschiedenen Modellierungsebenen. Nicht gezeigt ist jedoch, dass auch Modelle innerhalb einer Ebene zueinander in bestimmten Beziehungen stehen können.

Die Abbildung 2.7 verdeutlich beispielhaft die besagten Bezüge zwischen den verschiedenen UML-Modellarten Klassendiagramm, Objektdiagramm und Sequenzdiagramm. Im Klassendiagramm befinden sich die zwei Klassen Bibliothek und Buch. Im Objektdiagramm befinden sich zu den Klassen korrespondierende Modellelemente. Diese stellen Objektinstanzen der Klassen dar,

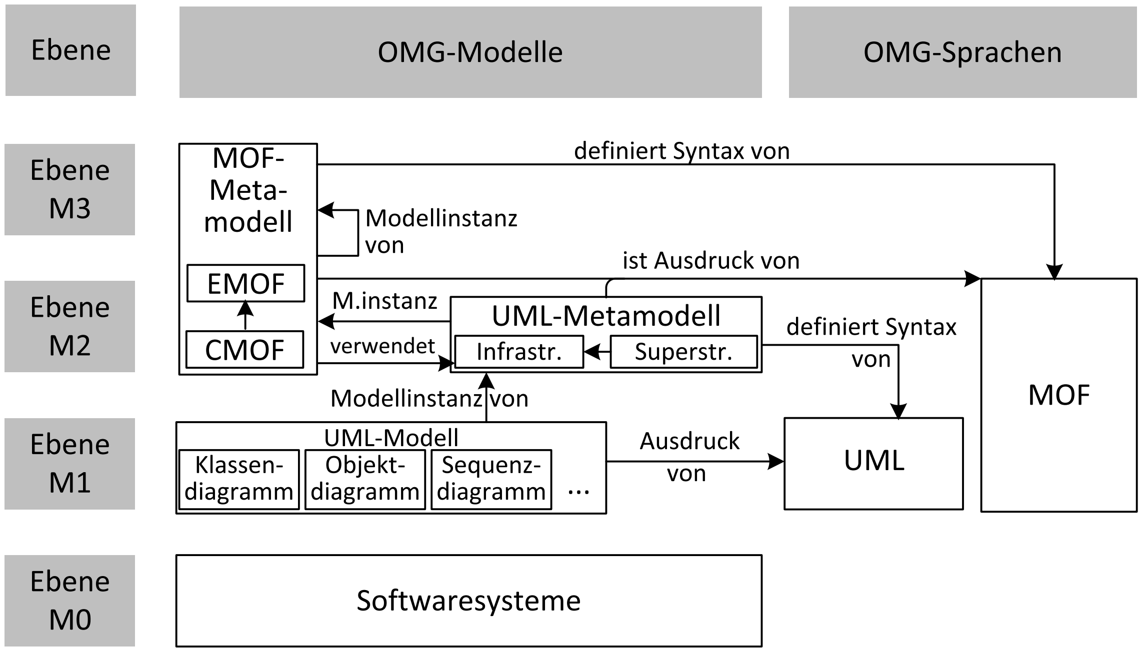


Abbildung 2.6: OMG-Schichtenmodell

was insbesondere durch die mit «instanceOf» benannte Beziehung dargelegt ist, die selbst aber kein Bestandteil von Klassen- oder Objektdiagrammen ist. Die Instanzbeziehung zwischen Objektinstanzen und Klassen wird daher auch über die Angabe des Klassennamens in den Objektinstanzen hergestellt. Sie ist ein Beispiel für einen *Querbezug* zwischen verschiedenen Modellarten.

In Objekt- und Sequenzdiagrammen können identische Teile des Originals modelliert werden. Hierbei werden allerdings unterschiedliche Attribute des Originals betont; in Objektdiagrammen wird die Objektstruktur (z.B. Objektattributbelegung und Referenzierungsstruktur) betont, in Sequenzdiagrammen die Aufrufreihenfolge zwischen Operationen verschiedener Objekte. Objekt- und Sequenzdiagramme können also verschiedene *Sichten* auf dieselben Originalteile darstellen. Die Tatsache, dass identische Originalteile in beiden Modellarten modelliert werden, führt dazu, dass Objektinstanzen in beiden Modellarten vorkommen. Zwischen einem Objekt- und einem Sequenzdiagramm kann also eine gewisse *Überlappung* herrschen.

Die in dieser Arbeit entwickelten und in Abschnitt 2.2ff eingeführten Prozessmodelle ordnen sich ebenfalls in einem 4-Ebenen-Metamodellierungsmodell ein. Vorhandene Gemeinsamkeiten zwischen den Prozessmodellen finden sich dabei auch als Überlappungen wieder. Die Querbezüge zwischen den Prozessmodellen sind wesentlich für die Realisierung bestimmter Modellprüfungen, worauf in Kapitel 4 näher eingegangen wird.

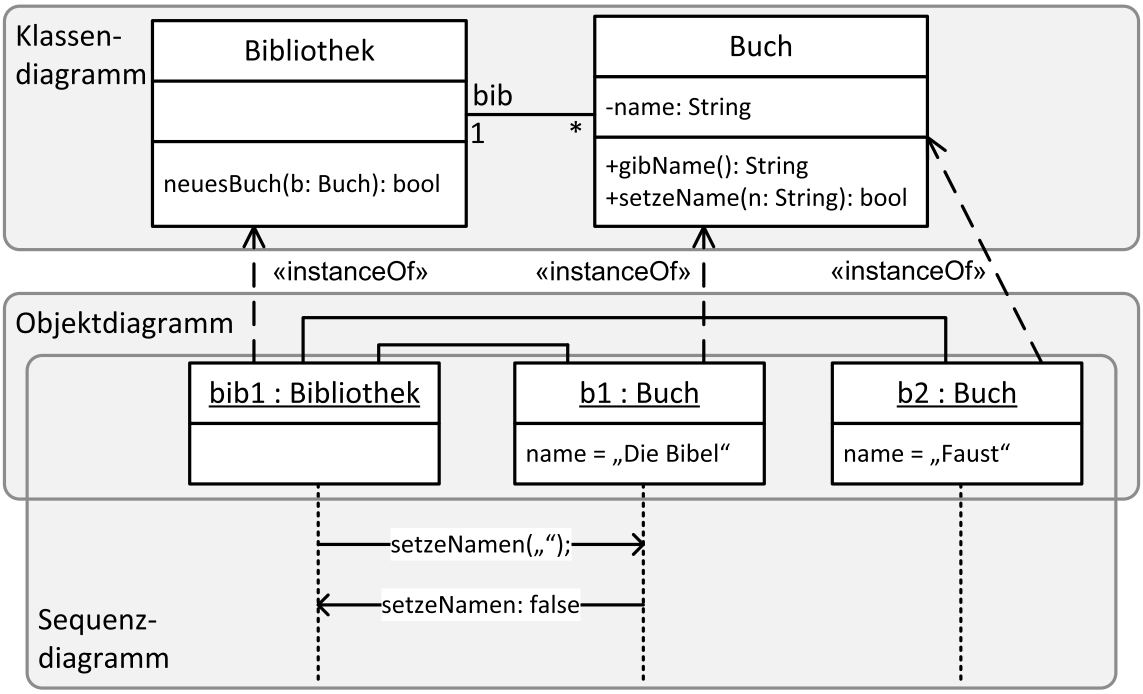


Abbildung 2.7: Querbezüge und Überlappungen in UML-Diagrammen

2.1.5 Standards für Prozessmodellierungssprachen

Es existiert eine Reihe von Sprachen, in denen Modelle von Prozesstypen ausgedrückt werden können. Die Wahl der Sprache richtet sich in erster Linie nach dem beabsichtigten Modellzweck. Für die fachliche Analyse bestehender Geschäftsprozesstypen werden beispielsweise UML-Aktivitätsdiagramme oder die BPMN verwendet. Modelle in diesen Sprachen abstrahieren noch von technischen Details und eignen sich daher nicht für die direkte Ausführung in Prozessmanagementsystemen im Gegensatz zu Modellen in WS-BPEL oder XPDL.

Modellierungszeit-Flexibilität

Prozessdefinitionsmodelle modellieren nicht ein einziges mögliches Prozessoriginal sondern einen Prozesstyp und somit eine Menge von Prozessen. Diese Eigenschaft kann durch die *Modellierungszeit-Flexibilität* der verwendeten Sprache gefördert werden. Modellierungszeit-Flexibilität bezieht sich auf Elemente der Sprache, die Prozessvarianten zulassen. Beispielsweise gibt es in BPMN oder WS-BPEL Sprachelemente, mit denen Aktivitätsalternativen (Verzweigungen) oder Iterationen (Schleifen) modelliert werden können. Prozessmodelle, in denen diese Elemente vorkommen, modellieren dann Prozesse, in denen die betreffenden Aktivitäten alternativ oder wiederholt durchgeführt werden.

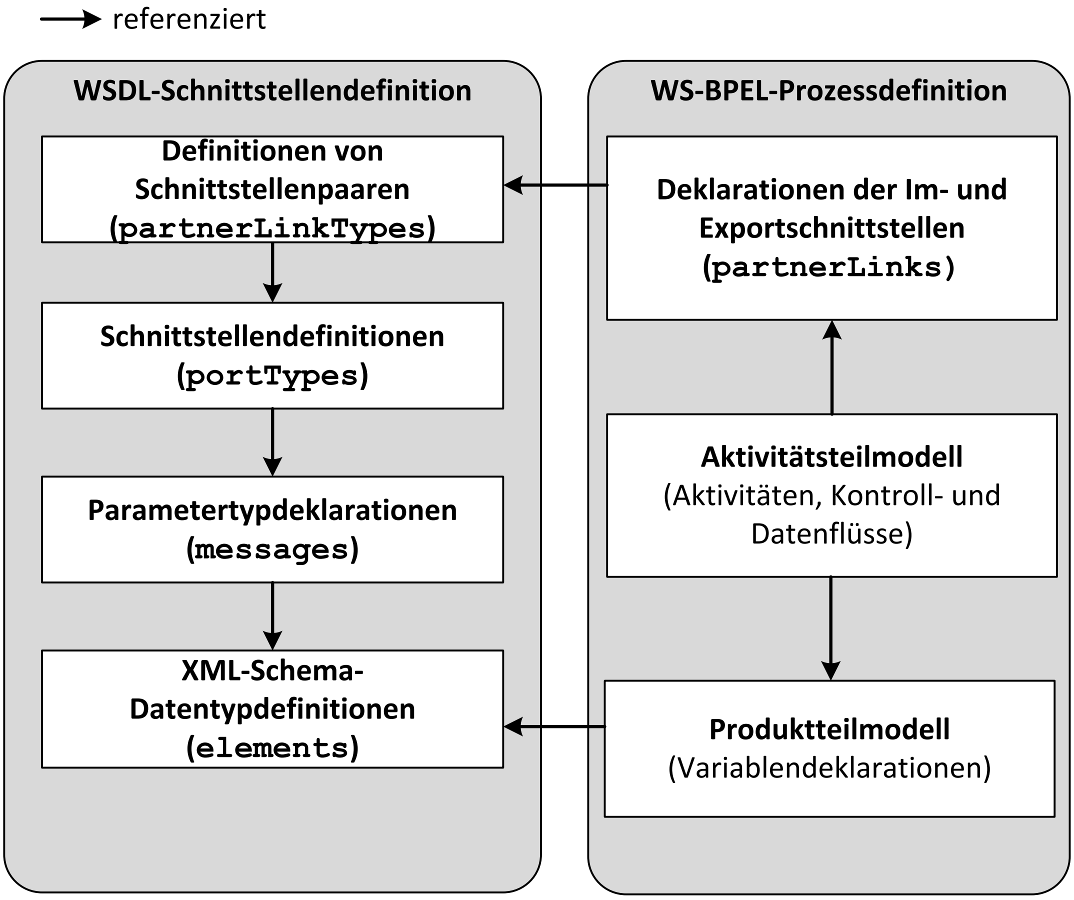


Abbildung 2.8: Aufbau von WS-BPEL-Dokumenten

WS-BPEL

Die für diese Arbeit zentrale Prozessmodellierungssprache ist die *Web Service Business Process Execution Language (WS-BPEL)* in der Version 1.1. In WS-BPEL können Prozessdefinitionsmodelle so präzise modelliert werden, dass sie automatisch interpretierbar und daher in einer Prozess-Laufzeitumgebung ausführbar sind. In [AAA^{+}07] sind gültige Ausdrücke der Sprache beschrieben. Hierin wird insbesondere festgelegt, dass WS-BPEL-Modelle als XML-Dokumente, im Folgenden WS-BPEL-Dokumente genannt, zu serialisieren sind. Die Semantik der einzelnen Sprachelemente, deren textuelle Syntax über die entsprechende XML-Serialisierung definiert wird, ist nur informal festgehalten. Einige Forschungsansätze wie z.B. [OVA^{+}07, LM07, VA05] versuchen, die Semantik von WS-BPEL durch Abbildung auf verschiedene Formalismen zu formalisieren.

WSDL Auf technischer Seite verhalten sich WS-BPEL-Prozesse wie WebServices und stützen sich umgekehrt auf andere WebServices ab. Daraus folgt insbesondere, dass WS-BPEL-Prozessdefinitionsmodelle Schnittstellen referenzieren. Die Definition der Schnittstellen (portTypes) geschieht in separaten XML-Dokumenten, die dem Standard *Web Service Description Language (WSDL)* genügen. WSDL-Schnittstellendefinition legen insbesondere Funktionssignaturen fest, d.h. Funktionsnamen sowie Ein- und Ausgabeparameter und deren Datentypen (messages). Die Definition der Parameterdatentypen kann wiederum in externen XML-Schema-Dokumenten erfolgen. Schnittstellen können zu Paaren zusammengefasst werden (partnerLinkTypes). Schnittstellenpaare (X, Y) werden bei asynchronen Funktionsaufrufe zwischen einem WebServices A, der X implementiert, und B, der Y implementiert, verwendet. Ruft A eine Funktion $Y.f$ von B asynchron auf, hält der Kontrollfluss in A nicht bis ein Rückgabewert eintrifft. Stattdessen liefert der aufgerufene Prozess B den Rückgabewert über eine Callback-Funktion $X.c$ zurück an A.

Aufbau von WS-BPEL WS-BPEL-Dokumente sind intern im Groben aufgebaut, wie in Abbildung 2.8 gezeigt. Offensichtlich ähneln diese Dokumente im Aufbau Programmen klassischer imperativer Programmiersprachen. In dem Sinne implementieren WS-BPEL-Prozessdefinitionsmodelle eine Menge von *Exportschnittstellen*. Im ersten Teil werden die Deklarationen der Exportschnittstellen angegeben. Der Aufruf von Exportschnittstellenfunktion kann zur Laufzeit die Erzeugung einer neuen Prozessinstanz bewirken oder eine wartende Prozessinstanz reaktivieren und somit den Prozessablauf vorantreiben. Über die Parameter der jeweiligen Funktionen können der Prozessinstanz externe Daten zugeführt werden. Gleichsam werden *Importschnittstellen* deklariert. Dies ist notwendig, wenn bestimmte Prozessaktivitäten über externe WebServices realisiert sind, die diese Schnittstellen exportieren. In einer solchen Aktivität wird dann eine Funktion einer Importschnittstelle unter Angabe bestimmter Aktualparameter referenziert; die Aktivität wird also durch einen Funktionsaufruf eines externen WebService realisiert. Die Deklaration der Im- und Exportschnittstellen im WS-BPEL-Dokument geschieht generell indirekt durch Referenzierung der oben angesprochenen `partnerLinksTypes` und nicht der `portTypes`.

In einem weiteren Abschnitt eines WS-BPEL-Dokuments können lokale Prozessvariablen deklariert werden. Die Deklaration referenziert dabei Datentypdefinitionen aus einem WSDL-Dokument. In Hinblick auf die in Unterabschnitt 2.1.3 beschriebenen Prozessteilmodelle kann dieser Abschnitt als Produktmodell gesehen werden. Es sei jedoch angemerkt, dass die “Produkte” hier weniger komplexe technische Dokumente darstellen, sondern Daten elementarer Typen wie Zeichenketten, Ganzzahlen oder Felder und Verbunde aus diesen.

Gegenstand der Betrachtungen in dieser Arbeit ist das Aktivitätsteilmodell eines WS-BPEL-Prozessdefinitionsmodells. Hierin finden sich die Prozessschritte und Kontrollflussdefinitionen, die festlegen, in welcher Reihenfolge welche Teilmengen der Aktivitäten durchgeführt werden können.

Typisch für Aktivitätsteilmodelle in WS-BPEL-Prozessdefinitionsmodellen ist ihr hierarchischer Aufbau. Tatsächliche Prozessschritte werden in atomaren Aktivitäten modelliert, bestimmte Kontrollflussdefinitionen über komplexe Aktivitäten, die weitere Aktivitäten enthalten können. Die wichtigsten WS-BPEL-Aktivitätstypen sind dabei:

➢ *Atomare Aktivitäten*

  – `Invoke`. Eine `Invoke`-Aktivität steht für einen synchronen oder asynchronen Aufruf eines externen WebServices; eine `Invoke`-Aktivität ist also immer der *Aufruferseite* zugeordnet. Dieser WebService kann vollautomatisch durchgeführt oder durch einen menschlichen Prozessbeteiligten realisiert werden bzw. wiederum durch einen anderen WS-BPEL-Prozess durchgeführt werden. In einer solchen Aktivität werden die aufgerufene Importschnittstellenfunktion sowie zugehörige Aktualparameter angegeben.

  – `Receive`. Eine `Receive`-Aktivität markiert in einem aufgerufenen Prozess B im Normalfall den Startpunkt eines Kontrollflusses, ist also der Seite eines *aufgerufenen Prozesses* zuzuordnen. Sie referenziert eine Funktion $X.o$ in der Exportschnittstelle eines WS-BPEL-Prozessdefinitionsmodells X. Wird $X.o$ aufgerufen, so wird eine Prozessinstanz erzeugt, deren Kontrollfluss bei der zugehörigen `Receive`-Aktivität beginnt. Eventuelle Eingabeparameter werden gemäß Angaben innerhalb der `Receive`-Aktivität in einer lokalen Prozessvariablen gespeichert.

  – `Reply`. Das Gegenstück einer `Receive`-Aktivität ist eine `Reply`-Aktivität. In ihr endet normalerweise der Kontrollfluss auf Seiten eines *aufgerufenen Prozesses*. Eventuelle Ausgabeparameter werden analog gemäß Angaben innerhalb der `Reply`-Aktivität aus einer lokalen Prozessvariablen entnommen.

  Weitere atomare Aktivitäten sind `Throw`, `Wait`, `Empty`, `Compensate` und `Assign`. Diese sind für diese Arbeit nicht weiter wichtig und werden daher an dieser Stelle nicht weiter erläutert.

➢ *Komplexe Aktivitäten* fassen (komplexe und atomare) Aktivitäten zusammen, beinhalten also andere Aktivitäten. Durch sie wird innerhalb des Aktivitätsmodells eine Hierarchie erzeugt. Dieser Hierarchie ist vergleichbar mit der Blockstruktur-Hierarchie in imperativen Programmiersprachen. Blöcke entsprechen dabei komplexen Aktivitäten, Anweisungen atomaren Aktivitäten. Die Enthaltenseinsbeziehungen zwischen Aktivitäten wird innerhalb des Aktivitätsabschnitts der WS-BPEL-Definition in die Enthaltenseinsbeziehungen der entsprechenden XML-Elemente abgebildet. Blöcke in imperativen Programmiersprachen werden im Allgemeinen zur Begrenzung des Gültigkeitsbereichs bestimmter Kontrollstrukturen wie Verzweigungen und Schleifen verwendet. Gleiches gilt auch für komplexe Aktivitäten. Hiervon gibt es wiederum verschiedene, von denen die in dieser Arbeit verwendeten im Folgenden kurz erläutert werden.

- `Sequence`. Die in einer `Sequence`-Aktivität enthaltenen Aktivitäten werden in einer linearen Reihenfolge durchgeführt. Diese Reihenfolge wird dabei durch die Reihenfolge der entsprechenden XML-Kindelemente im betreffenden WS-BPEL-Dokument bestimmt.
- `Flow / Link`. Die in einer `Flow`-Aktivität enthaltenen Aktivitäten können in einer beliebigen Reihenfolge ausgeführt werden. Insbesondere ist daher auch die Reihenfolge der XML-Kindelemente im betreffenden WS-BPEL-Dokument unerheblich. Die Reihenfolge kann wiederum durch `Links` zwischen Aktivitäten eingeschränkt werden. Links besitzen stets eine Quell- und eine Zielaktivität, wobei die Zielaktivität erst dann ausgeführt werden kann, wenn die Quellaktivität bereits ausgeführt wurde.
- `Switch / Case`. Eine `Switch`-Aktivität hat `Case`-Aktivitäten als Kinder. Jede `Case`-Aktivität besitzt eine Bedingung, d.h. einen wahrheitswertigen Ausdruck. Zur Prozesslaufzeit wird das erste `Case`-Kind ausgeführt, dessen Bedingung zutrifft. Die Ausführung einer `Case`-Aktivität bewirkt, dass ihre einzige (komplexe) Kindaktivität ausgeführt wird.
- `While`. Das Kind einer `While`-Aktivität wird solange ausgeführt wie die der `While`-Aktivität zugeordnete Bedingung wahr ist.

Eine weitere komplexe Aktivität ist die `Pick`-Aktivität, die im Folgenden aber keine Rolle spielt.

Viele der über komplexe Aktivitäten umgesetzten Kontrollflusskonstrukte sind klassischen imperativen Programmiersprachen entlehnt. Hierdurch hat WS-BPEL einen stark imperativen Charakter. Unterschiede zu diesen Programmiersprachen finden sich in der Umsetzung von Nebenläufigkeit wie sie in der `Flow`-Aktivität zu finden ist.

Mit Links können komplizierte, graphische Kontrollflussdefinitionen vorgenommen werden. Ein Link wird “aktiviert”, sobald seine Quellaktivität abgeschlossen wurde und eine eventuelle vorhandene “Wächterbedingung” (`transitionCondition`) zutrifft. Eine Aktivität kann ausgeführt werden, wenn eine Bedingung `joinCondition` zutrifft, die ein boolescher Ausdruck ist, der sich auf Aktivierungszustände eingehender Links bezieht. Beispielsweise kann

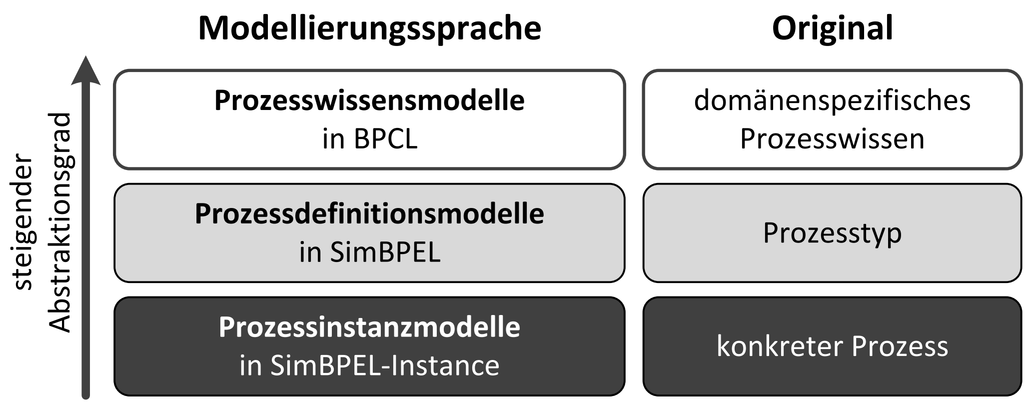


Abbildung 2.9: Prozessmodell-Abstraktionsebenen in dieser Arbeit

über eine boolesche Disjunktion eine Aktivität bereits ausgeführt werden, wenn nur eine (bzgl. Link-Verbindung) von mehreren Vorgängeraktivitäten abgeschlossen wurde. In dieser Arbeit wird, sofern nicht anders angegeben, jedoch folgendes Verhalten angenommen: Eine Aktivität kann erst dann durchgeführt werden, wenn alle ihre Vorgängeraktivitäten abgeschlossen wurden. Soweit nicht anders angegeben, haben Links keine Wächterbedingungen.

Die Syntax von WS-BPEL ist in [AAA$^+$07] über eine XML-Darstellung erklärt. XML-Dokumente lassen sich technisch einfach verarbeiten, sind für einen menschlichen Betrachter jedoch schwer zu verstehen, insbesondere dann, wenn in ihnen komplizierte Link-Strukturen verwendet werden.

In den folgenden Abschnitten wird als erstes grundlegendes Ergebnis dieser Arbeit, verschiedene Prozessmodellierungssprachen auf unterschiedlichen Abstraktionsebenen erläutert. WS-BPEL bildet dabei den Ausgangspunkt einer graphischen Sprache für Prozessdefinitionsmodelle, die die mittlere von drei Abstraktionsebenen bildet.

2.2 Prozessmodell-Abstraktionsebenen

Prozessmodelle sind das Kommunikationsmittel zwischen Prozessmanagementsystem, Prozessbeteiligten und Prozessmodellierer. Mit Prozessmanagementsystemen werden im Allgemeinen Modellierungssprachen für Prozessdefinitionsmodelle verbunden. Prozessdefinitionsmodelle eignen sich für die Modellierung von Prozesstypen. Sie sind jedoch zu abstrakt zur Modellierung konkreter Prozesse, da sie von konkreten Ausführungszuständen abstrahieren. Andererseits sind sie zu detailliert und damit zu restriktiv zur Modellierung domänenspezifischen Prozesswissens.

Im Folgenden werden drei Modellierungssprachen zur Modellierung von Prozessen auf unterschiedlichem Abstraktionsniveau erläutert. Die Abbildung 2.9

stellt diese einander hinsichtlich des Abstraktionsgrads ihrer Modelle und der modellierten Originale gegenüber. Konkrete Prozesse, wie beispielsweise die Abwicklung eines bestimmten Schadensfalls eines bestimmten Versicherungsnehmers, werden in Prozessinstanzmodellen modelliert, wofür die so genannte SimBPEL-Instance als Sprache verwendet wird. Prozesstypen, die von konkreten Prozessen abstrahieren, werden in Prozessdefinitionsmodellen mittels SimBPEL modelliert. In Prozessdefinitionsmodellen finden sich durch den Abstraktionsschritt insbesondere keine Ausführungszustände wie in Prozessinstanzmodellen. Domänenspezifisches Prozesswissen gilt prozesstypübergreifend und bezeichnet in dieser Arbeit Wissen über Ausführungshäufigkeiten und -reihenfolgen von Aktivitäten, die nicht spezifisch für einen Prozesstyp sind, sondern allgemein in einer Anwendungsdomäne gelten. Prozesswissen wird in Prozesswissensmodellen modelliert. Die dafür entwickelte Sprache wird Business Process Compliance Language (BPCL) genannt.

2.2.1 Prozessinstanzmodelle

Das Original eines Prozessinstanzmodells ist ein konkreter Prozess. Prozessinstanzmodelle dienen in erster Linie der Kommunikation zwischen Prozessbeteiligtem und einem Prozessmanagementsystem. Sie spiegeln einerseits Informationen über einen laufenden Prozess wider, d.h. über sie kommuniziert ein Prozessmanagementsystem Informationen über einen Prozess an einen Prozessbeteiligten. Andererseits können Sie manipuliert werden, was sich wiederum auf die im Prozessmanagementsystem vorgehaltenen Prozesszustandsinformationen niederschlägt, d.h. ein Prozessbeteiligter kann über Prozessinstanzmodelle Prozessfortschritte wie den Abschluss einer Aktivität an das Prozessmanagementsystem kommunizieren.

Übliche, tabellarische Prozessinstanzmodelle

Üblicherweise werden Prozessinstanzen in Prozessmanagementsystemen tabellarisch dargestellt. Solche Tabellen kann man als Prozessinstanzmodelle auffassen. In ihnen wird der Zustand eines Prozesses durch tabellarische Auflistung enthaltener Aktivitäten und deren aktuellen Zuständen widergespiegelt. Insbesondere die durch den WebSphere Process Server unterstützten Prozesse werden auf diese Weise modelliert und einem Prozessbeteiligten kommuniziert.

Prozessinstanzmodelle mit rein tabellarischer Syntax sind ungeeignet für den Umgang mit Dynamiksituationen in Prozessen. Aus ihnen geht zwar der aktuelle Prozesszustand hervor, jedoch nicht die Ablaufstruktur, also die Gesamtheit aller Aktivitäten mit der zugehörigen Kontrollflussdefinition, wie sie üblicherweise in Prozessdefinitionsmodellen modelliert ist. Dynamische

Änderungen beziehen sich jedoch gerade auf die Ablaufstruktur. Diese muss deshalb einem Prozessbeteiligtem zugänglich sein, um dynamische Prozessänderungen vornehmen zu können.

Diagrammatische Prozessinstanzmodelle

In dieser Arbeit wurde eine diagrammatische Sprache für Prozessinstanzmodelle entwickelt. Wie auch die üblichen tabellarischen Prozessinstanzmodelle spiegeln diese Prozessinstanzmodelle den Ausführungszustand eines bestimmten Prozesses wider. Hinzu kommt jedoch die Ablaufstruktur, die identisch mit einem vorhandenen Prozessdefinitionsmodell sein mag, jedoch nicht sein muss. Letzteres ist der Fall, wenn dynamische Änderungen durchgeführt wurden. Diese werden gerade über Prozessinstanzmodelle vermittelt, d.h., dass ein Prozessbeteiligter in der Lage ist, Modifikationen an der Ablaufstruktur innerhalb eines Prozessinstanzmodells vorzunehmen. Diese Modifikationen werden dann an das ausführende Prozessmanagementsystem propagiert.

Abbildung 2.10 zeigt ein Beispiel eines Prozessinstanzmodells in der konkreten, diagrammatischen Syntax, wie sie in den folgenden Kapiteln und auch der prototypisch entwickelten Werkzeugunterstützung verwendet wird. Die Sprachelemente entsprechen teils denen der Modellierungssprache WS-BPEL für Prozessdefinitionsmodelle, wie in Unterabschnitt 2.1.5 beschrieben:

Aktivitäten sind mehrfach beschriftet. Das generelle Beschriftungsschema ist wie folgt: Der `BPEL-Typ` steht an oberster Position, links daneben das zugehörige Icon. Fettgedruckt ist der Aktivitätsname, abgetrennt durch ein Doppelpunkt und schräggestellt der *Aktivitätstyp*. Letzterer bekommt durch die in Unterabschnitt 2.2.3 beschriebenen Prozesswissensmodelle seine Bedeutung.

Prozessvariablen (`AtomicDatum`) werden explizit dargestellt. Sie sind mit Aktivitäten über Schreibzugriffs- (`writesTo`) und Lesezugriffsbeziehungen (`readsFrom`) verbunden.

Hinzu kommen Elemente, die den Ausführungszustand von Aktivitäten modellieren: Aktivitätsinstanzen (`ActivityInstance`) werden als Rechtecke innerhalb von `Invoke`-Aktivitäten notiert. Sie sind mit Active oder Finished beschriftet. Fehlende `ActivityInstances` innerhalb einer `Invoke`-Aktivität zeigen an, dass sich die Aktivität im Zustand `Waiting` bzw. `Dead` befindet und ggf. aufgrund der Kontrollflussdefinition noch nicht bzw. nicht mehr aktiviert werden kann. Aktivitäten innerhalb von `While`-Elementen können ggf. mehrfach durchgeführt werden. Eine mehrfache Ausführung wird in Prozessinstanzmodellen so modelliert, dass sich in den betreffenden Aktivitäten mehrere übereinanderliegende `ActivityInstances` wiederfinden. Hierbei gehört die oberste zur ersten, die unterste zur zuletzt bzw. aktuell ausgeführten

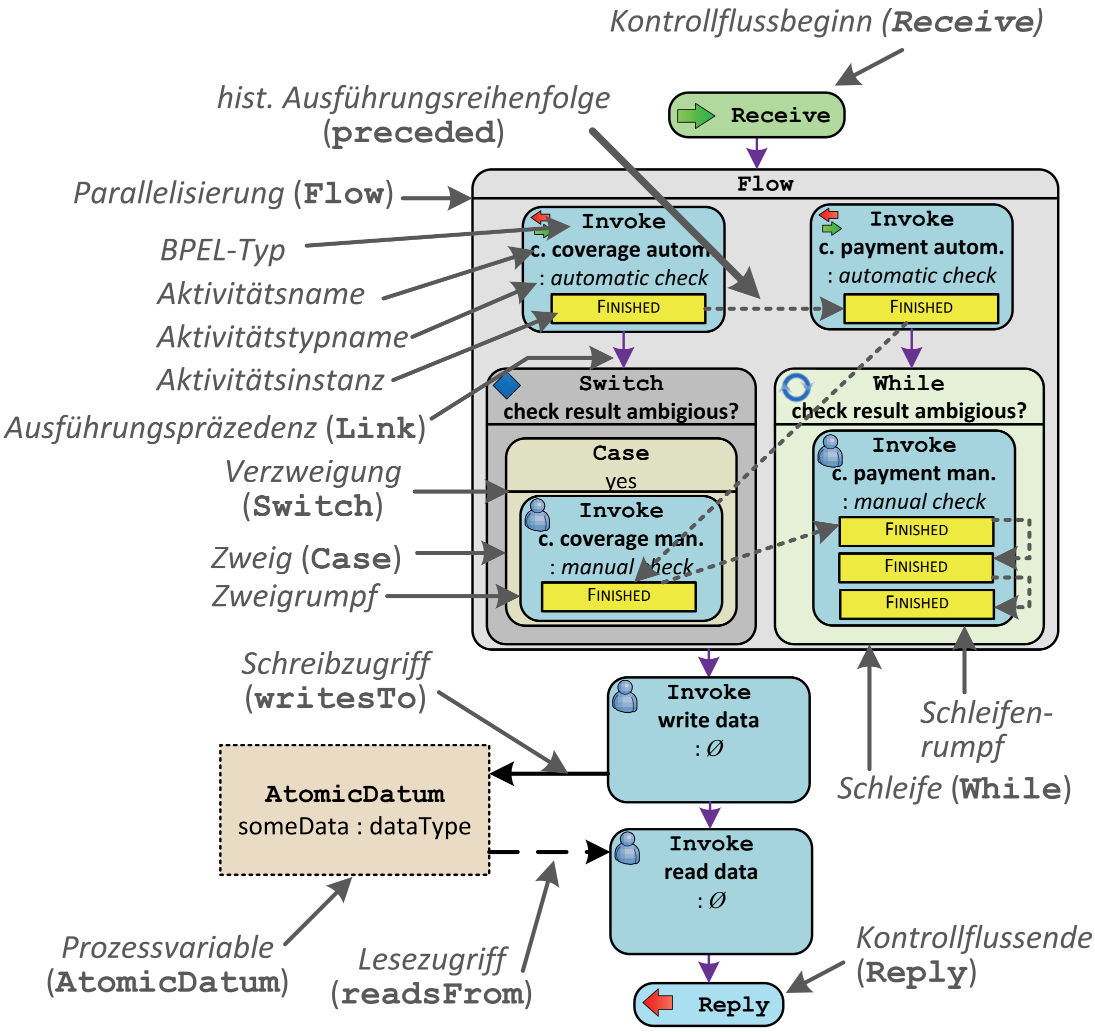


Abbildung 2.10: Syntaktische Elemente eines Prozessinstanzmodells

Iteration.

Aktivitätsinstanzen sind über preceded-Referenzen (strichlierte Pfeile) verbunden. Die Verkettung gibt die tatsächliche historische Ausführungsreihenfolge bereits durchgeführter Aktivitäten an, d.h. die Reihenfolge, in denen die Aktivitätsinstanzen den Zustand Finished angenommen haben. Im Prozessinstanzmodell von Abbildung 2.10 ist beispielsweise ablesbar, dass check coverage automatically vor check payment automatically finalisiert wurde.

Die Verkettung über preceded-Referenzen ergibt immer eine lineare Liste. Dies ist auch dann der Fall, wenn Aktivitäten parallel durchgeführt werden können, wie beispielsweise check coverage automatically und check payment automatically. Auch in diesen Fällen wird eine der zugehörigen Aktivitätsinstanzen im jeweiligen Prozess zuerst finalisiert, im Fall von Abbildung 2.10 check coverage automatically.

2.2.2 Prozessdefinitionsmodelle

Prozessdefinitionsmodelle sind mit Prozessinstanzmodellen eng verwandt. Ihre Syntax ist nahezu identisch, jedoch fehlen hier Informationen, die spezifisch für einen bestimmten Prozess sind, da Prozessdefinitionsmodelle Prozesstypen modellieren. Die prozessspezifischen Informationen sind Fortschrittsinformationen, also die `ActivityInstances` zusammen mit den preceded-Referenzen. Die Auslassung dieser Modelldetails entspricht dem Abstraktionsschritt zwischen Prozess und Prozesstyp.

2.2.3 Prozesswissensmodelle

Der Abstraktionsschritt zwischen Prozessinstanz- und -definitionsmodell ist recht gering. Der Grund hierfür ist, dass Prozessdefinitionsmodelle immer noch detailliert genug sein müssen, um von einem Prozessmanagementsystem interpretiert werden zu können und somit zur Unterstützung von Prozessen eines bestimmten Typs dienen können.

Prozessdefinitionsmodelle sind daher ungeeignet zur Modellierung allgemeinen Prozesswissens einer bestimmten Anwendungsdomäne wie die der Versicherungen. Es ist in ihnen nicht möglich, Prozesswissen ohne Bezug auf einen bestimmten Prozesstyp zu modellieren. Des Weiteren sind die Ausdruckmittel innerhalb von Prozessdefinitionsmodellen ungeeignet, um bestimmte Sachverhalte geeignet zu modellieren. Beispielsweise kann die Durchführungsabhängigkeit zweier Aktivitäten nicht ohne Festlegung ihrer Ausführungsreihenfolge erfolgen. Es ist also nicht möglich auszudrücken, dass im Falle der Durchführung einer Aktivität A auch eine Aktivität B *irgendwann* während der Prozessdurchführung durchgeführt werden muss. Ebenso ist es weder möglich etwas genauer auszudrücken, dass im Falle der Durchführung von A irgendwann *im Anschluss* B, noch dass B irgendwann *im Vorhinein* durchgeführt werden muss.

In dieser Arbeit wurde daher, zusätzlich zu den Sprachen für Prozessinstanz- und -definitionsmodelle, eine Sprache für Prozesswissensmodelle entwickelt, die abstrakter als Prozessdefinitionsmodelle sind. Prozesswissensmodelle abstrahieren von konkreten Kontrollflussdefinitionen und legen stattdessen nur fest, welche Typen von Aktivitäten wie häufig in bestimmten Prozessen vorkommen müssen bzw. welche (indirekten) Reihenfolgen Aktivitäten einhalten müssen oder welche Aktivitäten einander generell bedingen oder ausschließen.

Abbildung 2.11 zeigt ein Beispiel eines Prozesswissensmodells. Das Beispiel greift dabei einige Bedingungen auf, die für Schadenabwicklungsprozesse in Allgemeinen sinnvoll sind.

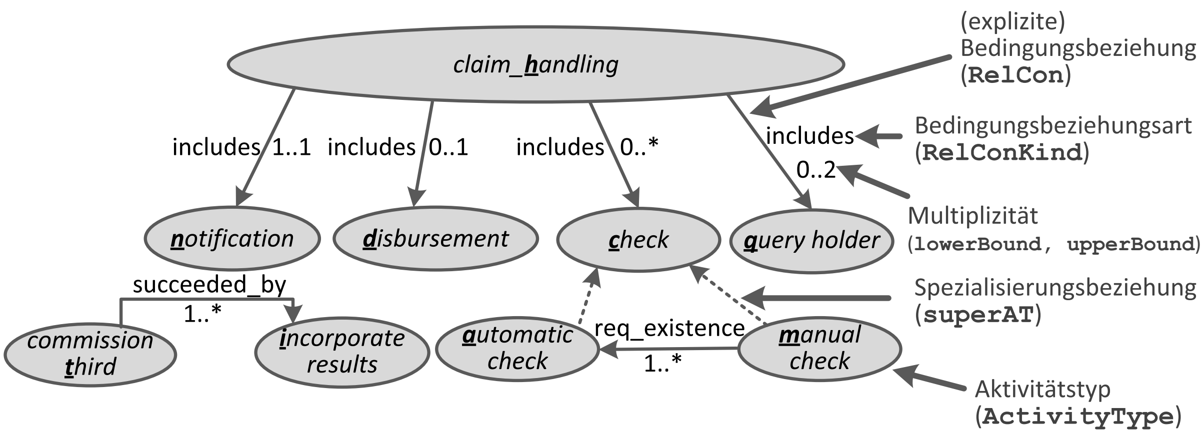


Abbildung 2.11: Beispiel eines Prozesswissensmodells

Aktivitätstypen

Ellipsen stellen hierbei *Aktivitätstypen* dar. Die Ellipsen sind beschriftet mit einem Aktivitätstypnamen. Aktivitätstypen repräsentieren Aktivitäten auf fachlicher Ebene und abstrahieren daher von jedweden Details wie beispielsweise Ein- und Ausgabeparameter von Aktivitäten. Wenn im Folgenden von einer *X*-Aktivität geschrieben wird, dann ist damit verkürzt eine Aktivität vom Aktivitätstyp mit Namen *X* gemeint.

Bedingungsbeziehungen

Aktivitätstypen können untereinander in bilaterale Beziehungen gesetzt werden. In den Prozesswissensmodellen wird dies durch Kanten zwischen den Aktivitätstypen ausgedrückt. Es werden hierbei zwei Arten von Beziehungen unterschieden: Bedingungsbeziehungen und Spezialisierungsbeziehungen.

Bedingungsbeziehungen werden in Prozesswissensmodellen als durchgezogene Pfeile notiert. Sie stellen jeweils immer eine Bedingung für den Quellaktivitätstyp dar. Vereinfacht gesprochen kann die Bedingung durch n-fache Durchführung einer Aktivität des Zielaktivitätstyps erfüllt werden. Eine Bedingungsbeziehung

$$A \xrightarrow[n..m]{\text{Art}} B$$

ist beschriftet mit dem Namen einer *Bedingungsbeziehungsart* (Art) und einer *Multiplizität* $n..m$, wobei $(n,m) \in \mathbb{N}_0 \times \mathbb{N} \cup \{*\}$ ist ($*$ steht für “unendlich”).

Bedingungsbeziehungsarten

Es werden folgende Bedingungsbeziehungsarten unterschieden:

Inklusion In Schadenabwicklungsprozessen sollte der Versicherungsnehmer über die Annahme oder Ablehnung der Schadensregulierung genau einmal unterrichtet werden. Keine Benachrichtigung würde gegen das berechtigte Interesse des Versicherungsnehmers über den Ausgang verstoßen, mehr als eine Benachrichtigung birgt die Gefahr widersprüchlicher Bescheide. Die Notwendigkeit der Durchführung einer entsprechenden Aktivität kann durch eine includes-Bedingungsbeziehung innerhalb von Prozesswissensmodellen ausgedrückt werden.

In diesem Fall besagt *claim_ handling* $\xrightarrow[1..1]{\text{includes}}$ *notification*, dass innerhalb eines Schadensabwicklungsprozesses die Benachrichtigung des Versicherungsnehmers genau einmal stattfinden muss. Die Schadensauszahlung hingegen soll gar nicht oder maximal einmal stattfinden. Dementsprechend ist *claim_handling* $\xrightarrow[0..1]{\text{includes}}$ *disbursement* modelliert. Anfragen an den Versicherungsnehmer sollen aus Gründen der Kundenzufriedenheit auf maximal zwei beschränkt sein, daher *claim_handling* $\xrightarrow[0..2]{\text{includes}}$ *query_holder*. Die includes-Bedingungsbeziehung von *claim handling* zu *check* hat eine unbeschränkte Multiplizität $0..*$. Sie stellt daher keine wirkliche Bedingung im Sinne einer Einschränkung dar, könnte daher ebenso gut ausgelassen werden. Allgemein bedeutet eine $b \xrightarrow[n..m]{\text{includes}} a$ Bedingungsbeziehung also, dass in einem b-Prozess eine a-Aktivität mindestens n- und maximal m-mal durchgeführt werden muss. Prozesse werden hierbei einfach als komplexe Aktivitäten angesehen.

Beispiel 2.3 (Inklusion)
Die Bedingungsbeziehung *claim_handling* $\xrightarrow[0..1]{\text{includes}}$ *disbursement* wird durch den im Prozessinstanzmodell aus Abbildung 1.12 modellierten Prozess verletzt. In diesem Prozessinstanzmodell sind zwei *disbursement*-Aktivitäten modelliert. Obwohl eine davon optional durchgeführt wird, ist die Bedingungsbeziehung verletzt, d.h. die Bedingungsbeziehung wird gewissermaßen pessimistisch gegenüber Prozessinstanz- und -definitionsmodellen evaluiert.

Existenz Mitunter muss eine Aktivität nicht unbedingt innerhalb eines bestimmten Prozesses durchgeführt werden, sondern in Abhängigkeit von der Durchführung einer anderen Aktivität. So ist es in Versicherungsprozessen beispielsweise sinnvoll, manuelle Prüfungen aus den zuvor bereits genannten Gründen durch entsprechende automatische Prüfungen zu ergänzen. Diese Durchführungsabhängigkeit wird innerhalb des Prozesswissensmodells durch die Bedingungsbeziehung *manual_check* $\xrightarrow[1..*]{\text{req\_existence}}$ *automatic_check* ausgedrückt. Diese besagt, dass, sofern eine *manual_check*-Aktivität ausgeführt wird, auch mindestens eine *automatic_check*-Aktivität ausgeführt werden muss. Allgemein ist unter einer Bedingungsbeziehung $b \xrightarrow[n..m]{\text{req\_existence}} a$ also zu verstehen, dass, sofern eine b-Aktivität in einem Prozess durchgeführt wird, mindestens n und maximal m a-Aktivitäten durchgeführt werden müssen. Insbesondere kann durch $b \xrightarrow[0..0]{\text{req\_existence}} a$ modelliert werden, dass keine a-Aktivität durchgeführt werden darf, sofern eine b-Aktivität durchgeführt wird. Die Belegung der oberen Schranke in der Multiplizitätsangabe mit 0 modelliert also einen *einseitigen Ausschluss*. Führt man hierzu noch die Bedingungsbeziehung $a \xrightarrow[0..0]{\text{req\_existence}} b$ ein, so ist insgesamt ein *wechselseitiger Ausschluss* zwischen b- und a-Aktivitäten modelliert.

Reihenfolge Existenz-Bedingungsbeziehungen treffen keine Aussage über zeitliche Ausführungsreihenfolgen von Aktivitäten. Derartige Bedingungen können über Bedingungsbeziehungen $b \xrightarrow[n..m]{\text{preceded\_by}} a$ bzw. $b \xrightarrow[n..m]{\text{succeeded\_by}} a$ in Prozesswissensmodellen modelliert werden. Erstere besagt, dass, falls eine b-Aktivität ausgeführt wird, *früher* im Prozess auch mindestens n und maximal m a-Aktivitäten ausgeführt werden müssen. Letztere besagt analog, dass, falls eine b-Aktivität ausgeführt wird, *später* im Prozess auch mindestens n und maximal m a-Aktivitäten ausgeführt werden müssen. In Versicherungsprozessen ist beispielsweise zu erwarten, dass bei Beauftragung Dritter (*commission third*) später ein Rücklauf zu erwarten ist, der ausgewertet und in den internen Versicherungsprozess eingepflegt werden muss. Daher ist im Prozesswissensmodell in Abbildung 2.11 modelliert, dass eine Beauftragung Dritter (*incorporate results*) auch immer mindestens eine Aktivität nach sich zieht, die den entsprechenden Rücklauf Dritter wieder einpflegt.

Spezialisierungsbeziehungen

Neben Bedingungsbeziehungen können in Prozesswissensmodellen auch Spezialisierungsbeziehungen zwischen Aktivitätstypen modelliert werden. Mittels Spezialisierungsbeziehungen können redundante Modellierungen von Bedingungsbeziehungen vermieden werden.

Wie zuvor beschrieben, bedeutet eine Bedingungsbeziehung $b \xrightarrow[n..m]{\text{Art}} a$, dass in einem Prozess bei Durchführung von b (Quelltyp) auch a (Zieltyp) durchgeführt werden muss (je nach *Art* ggf. vorher oder nachher). Ist nun zusätzlich $b' \xrightarrow{\text{specialization\_of}} b$ modelliert, so gilt implizit auch die Bedingungsbeziehung

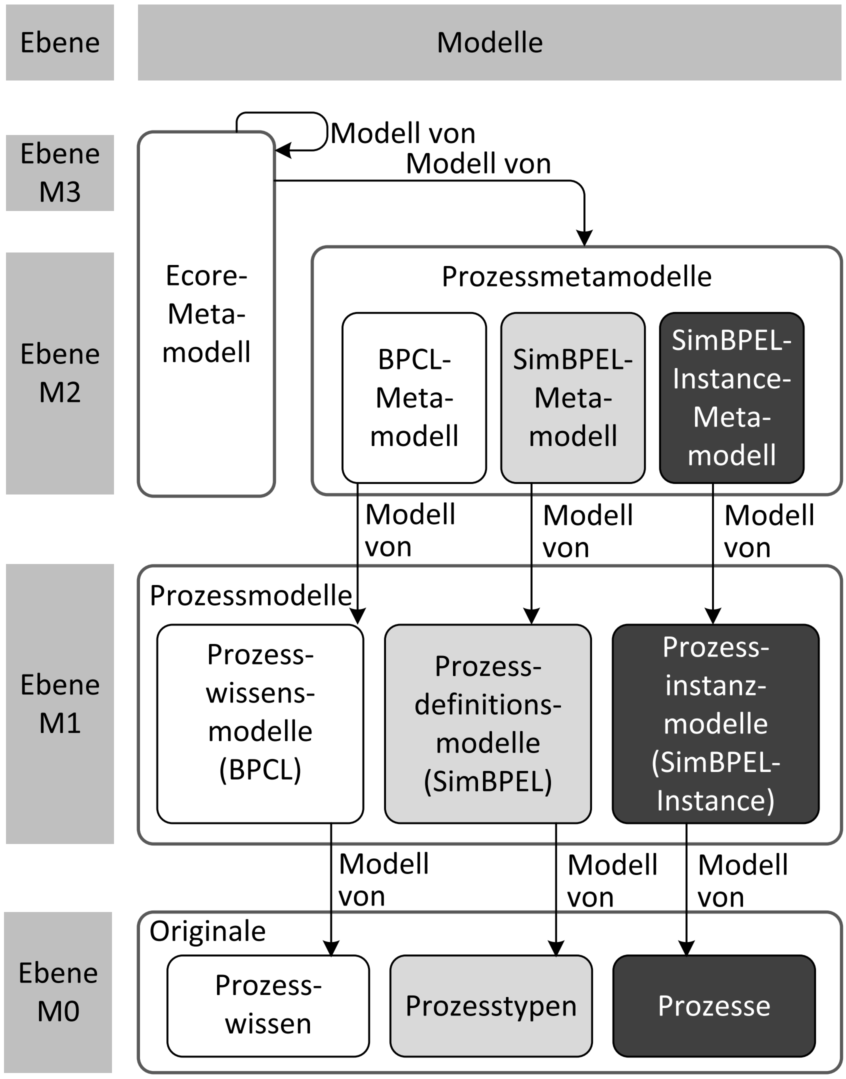


Abbildung 2.12: Meta-Ebenen für Prozessmodelle

$b' \xrightarrow[n..m]{\text{Art}} a$. Eine Spezialisierung eines Quelltyps führt also eine neue, implizite Bedingungsbeziehung ein. Der spezialisierende Typ (b') erbt gewissermaßen die Bedingungsbeziehungen seiner direkt oder indirekt spezialisierten Aktivitätstypen.

Ist des Weiteren $a' \xrightarrow{\text{specialization\_of}} a$ modelliert, so gelten Durchführungen von a'-Aktivitäten auch als Durchführungen von a-Aktivitäten. Vorkommnisse von a'-Aktivitäten in Prozessen können also zur Erfüllung oder Verletzung der Bedingungsbeziehung $b \xrightarrow[n..m]{\text{Art}} a$ beitragen.

2.3 Prozessmodellsyntax

Im vorhergehenden Abschnitt wurden Prozessmodelle auf unterschiedlichen Abstraktionsebenen nur exemplarisch erläutert. Die Syntax von Prozesswissensmodellen als auch von Prozessdefinitions- und -instanzmodellen ist jedoch formal definiert. Eine formale Syntaxdefinition ist notwendig zur späteren formalen Semantikdefinition der Modelle in Abschnitt 2.5. Zudem kann eine formale Syntaxdefinition zur Generierung von Modelleditoren verwendet werden. Generatoren hierfür existieren sowohl für textuelle Sprachen, beispielsweise MontiCore [KRV07, KRV08, GKR+08] als auch für die Kombination aus dem Eclipse Modeling Framework (EMF) [BSMP09] und dem Graphical Modeling Framework (GMF) [Gro09, Kapitel 4] für diagrammatische Sprachen. Die Rahmenwerke EMF und GMF werden in dieser Arbeit zur formalen Syntaxdefinition verwendet.

2.3.1 Abstrakte Syntax

Die Definition der Syntax der verschiedenen Prozessmodelle geschieht wie in Unterabschnitt 2.1.2 erläutert mittels Metamodellen. Die Prozessmetamodelle fügen sich in das Vier-Ebenen-Metamodellierungsmodell aus Unterabschnitt 2.1.4 ein, wie in Abbildung 2.12 gezeigt. In der Abbildung wird auf die explizite Angabe der jeweiligen Sprache verzichtet. Hier ist die Modell-von-Beziehung wieder so zu verstehen, dass die Quelle die Syntax einer Sprache definiert, wovon das oder die Ziele jeweils Ausdrücke sind.

Ecore

Ecore ist die Sprache, in der die Metamodelle der Prozessmodelle ausgedrückt sind und ist Bestandteil des Eclipse Modeling Frameworks. Ecore-Metamodelle ähneln EMOF-Metamodellen (vgl. Unterabschnitt 2.1.4). Gegenüber MOF-Metamodellen sind Ecore-Metamodelle somit einfacher. Beispielsweise sind Assoziationen zwischen Klassen, in Ecore EReferences genannt, immer gerichtet. Wie Abbildung 2.13 zeigt, rührt die Richtung daher, dass eine EReference immer einer EClass über die Referenz eReferences zugeordnet ist und eine zweite EClass über eReferenceType selbst nur referenziert.

Wie MOF ist Ecore reflexiv. Das Ecore-Metametamodell ist also selbst in Ecore ausgedrückt. Referenzen – notiert als Pfeile im Ecore-Metamodellteil aus Abbildung 2.13 – sind beispielsweise selbst wieder Modellinstanzen von EReference. Dieser Sachverhalt ist vor dem Hintergrund besonders vorteilhaft, dass aus Ecore-Metamodellen mittels des Eclipse Modeling Frameworks Editoren für Modelle generiert werden können, die den syntaktischen Festlegungen

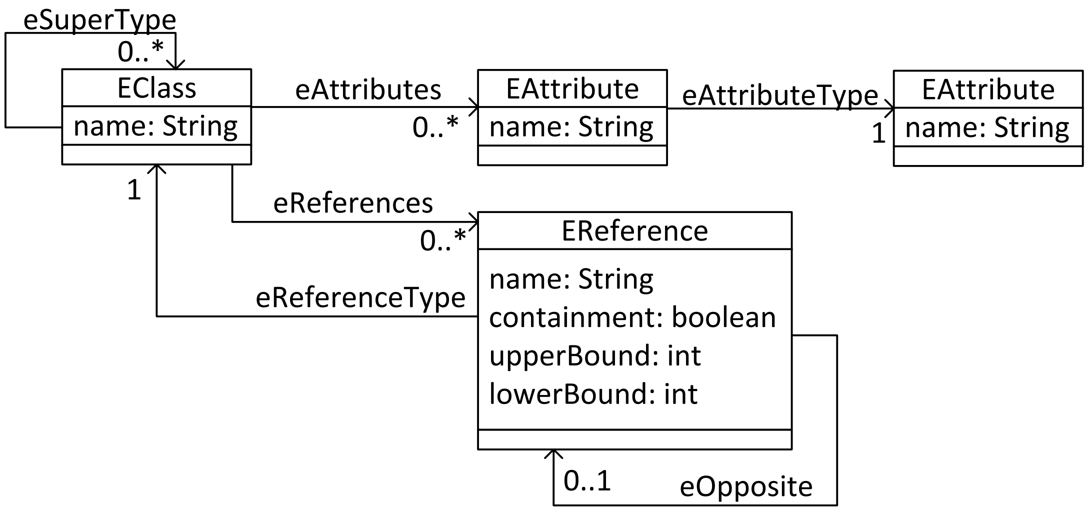


Abbildung 2.13: Kern des Ecore-Metametamodells [BSMP09, Seite 105]

der jeweiligen Metamodelle genügen. Durch die Reflexivität von Ecore kann auch ein Editor für Ecore-Metamodelle auf gleiche Weise gewonnen werden.

Sowohl das MOF-Metamodell, also auch das UML-Metamodell und UML-Klassenmodelle verwenden die gleiche konkrete Syntax. Diese wird im Allgemeinen als Klassendiagramm-Notation bezeichnet. Da Ecore und MOF Ähnliches leisten, liegt es nahe, auch Ecore-Metamodelle in der Klassendiagramm-Notation zu notieren. Dies ist der Fall in Abbildung 2.13 und auch bei allen weiteren Ecore-Metamodellen.

Prozessmetamodelle

Die abstrakte Syntax der in Abschnitt 2.2 erläuterten, unterschiedlich abstrakten Prozessmodelle ist in Prozessmetamodellen formal definiert. Ecore ist hierbei die Sprache, in denen die Prozessmetamodelle formuliert sind.

Abbildung 2.14 zeigt wesentliche Teile der Metamodelle aller drei Prozessmodellsprachen. Aus Gründen der Übersicht wurden einige Hilfsklassen weggelassen. Die Benennung der Schichten ist die der Prozessmodellsprachen. Prozesswissensmodelle werden in BPCL abgefasst, das für Business Process Compliance Language steht, zu deutsch: Geschäftsprozess-Komplianzsprache. Der Name deutet auf die spätere Verwendung dieser Modelle zur Prüfung fachlicher Bedingungen in Prozessdefinitions- und -instanzmodellen hin, die in Kapitel 4 erläutert wird. Prozessdefinitionsmodelle werden in SimBPEL modelliert. SimBPEL steht für “Simplified WS-BPEL”, also “vereinfachtes WS-BPEL”, und ist an WS-BPEL angelehnt, an einigen Stellen aber vereinfacht und an anderen Stellen erweitert. Prozessinstanzmodelle werden in SimBPEL-Instance ausgedrückt.

Gemeinsam ist Prozessmetamodellen und darüber hinaus allen Ecore-Metamodellen, dass Metaklassen in zwei voneinander unabhängigen Hierarchiebeziehungen zueinander stehen:

Generalisierung In Ecore können gemeinsame Merkmale, also gemeinsame Attribute und Referenzen, von Klassen $A_1, ..., A_n$ zu einer gemeinsamen Oberklasse B zusammengefasst werden. Zwischen den A_i und B besteht dann eine Generalisierungsbeziehung, d.h. B generalisiert die A_i. Besteht eine Generalisierungsbeziehung von A nach B, so weist A implizit alle Merkmale auf, die B (implizit) aufweist; man sagt, dass A von B *erbt*. Vererbung ist in objektorientierter Modellierung und Programmierung gängig und insoweit in Ecore nicht neu. Mehrfachvererbung ist in Ecore erlaubt, d.h. eine Klasse A kann von mehreren Klassen B_i erben; zyklische Vererbung ist verboten. Durch die mögliche Mehrfachvererbung induzieren Generalisierungsbeziehungen in einem Modell keinen Baum. In der konkreten Syntax der Metamodelle und in der Abbildung 2.14 ist eine Generalisierungsbeziehung wie aus UML-Klassendiagrammen bekannt durch einen Pfeil mit hohler Spitze notiert, der von der Unter- zur Oberklasse verläuft.

Komposition Kompositionsbeziehungen in Ecore-Metamodellen sind spezielle Referenzen und drücken Enthaltenseinsbeziehungen zwischen Modellelementen aus. Die Quelle einer Kompositionsbeziehung wird Aggregat genannt, das Ziel Teil. Jede Metaklasse ist Teil maximal eines Aggregats. Kompositionsbeziehungen sind azyklisch. Somit induzieren Kompositionsbeziehungen in einem Modell einen Wald von Hierarchien. Notiert werden Kompositionsbeziehungen wie aus der UML bekannt durch Linien zwischen Metaklassen, wobei das auf das Aggregat gerichtete Ende durch eine ausgefüllte Raute markiert ist.

BPCL-Metamodell Die Kompositionshierarchie im BPCL-Metamodell hat in `BpclModel` ihre Wurzel. Aktivitätstypen werden durch `ActivityType` repräsentiert. Diese können über die Referenz `superAT` einen oder keinen Supertyp referenzieren. Umgekehrt referenziert ein Aktivitätstyp über `subAT` seine Untertypen. Bedingungsbeziehungen zwischen Aktivitätstypen werden durch eine eigene Metaklasse `RelCon` repräsentiert. Über `target` bzw. `source` referenziert diese Metaklasse das Ziel bzw. die Quelle einer Bedingungsbeziehung. Entsprechende Referenzen in umgekehrter Richtung von `ActivityType` zu `RelCon` heißen `fromTarget` bzw. `fromSource`. Die Art der Bedingungsbeziehung wird in dem Attribut `conKind` gespeichert. Werte von `conKind` sind vom Enumerationstypen `RelConKind`. Die einer Bedingungsbeziehung zugeordne-

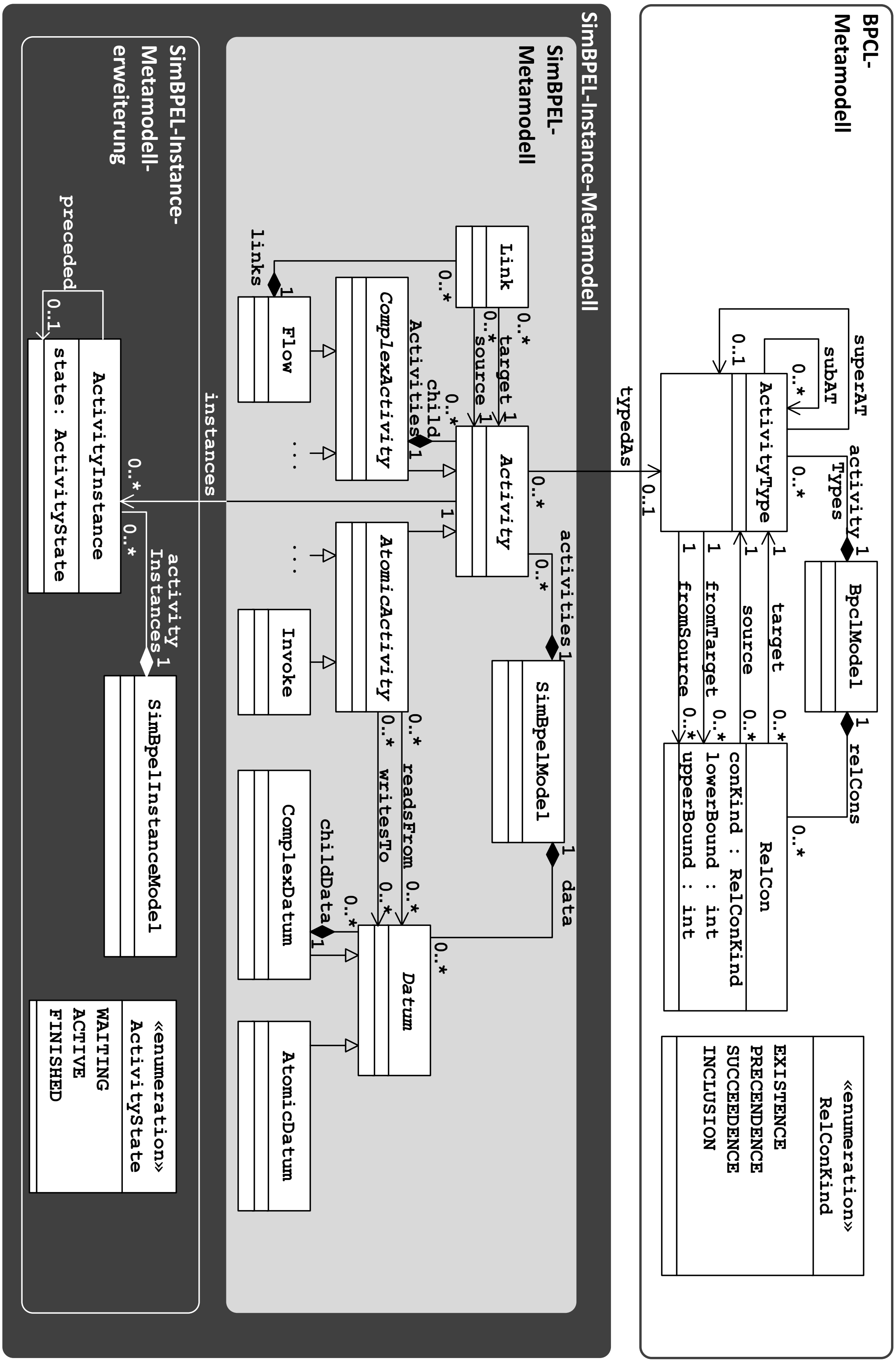


Abbildung 2.14: Verschränkte Prozessmetamodelle

ten Multiplizitäten werden in den Attributen `lowerBound` bzw. `upperBound` festgehalten.

SimBPEL-Metamodell Im SimBPEL-Metamodell ist eine vereinfachte, abstrakte Syntax von Prozessdefinitionsmodellen in WS-BPEL definiert. Zwar existiert eine Syntaxdefinition von WS-BPEL in Form eines XML-Schemas; dieses wird jedoch nicht verwendet aus folgenden Gründen: Erstens ist ein XML-Schema für die spätere Werkzeugunterstützung aus technischen Gründen ungeeignet. Zweitens ist das Schema mit Sprachdetails überladen, die für die spätere Werkzeugunterstützung irrelevant sind. Drittens fehlen gewisse Sprachbestandteile, wie beispielsweise explizite Datenflüsse. Für bestimmte Modellprüfungen sind explizite Datenflüsse jedoch erforderlich.

SimBPEL-Modelle bestehen aus Aktivitäten und Daten. Aktivitäten sind selbst in einer Hierarchie angeordnet. Komplexe Aktivitäten, die innere Knoten in der Hierarchie darstellen, werden durch Spezialisierungen von `ComplexActivity` repräsentiert, wie beispielsweise `Flow`-Aktivitäten (vgl. Unterabschnitt 2.1.5). Teile von komplexen Aktivitäten stehen mit ihrem Aggregat über die `childActivities`-Komposition in Beziehung. `Flow`-Aktivitäten beinhalten des Weiteren eine beliebige Anzahl `Links`, die jeweils eine Aktivität als `target` und eine als `source` referenzieren. Atomare Aktivitäten bilden die Blätter in der Kompositionshierarchie; sie haben keine weiteren Teile innerhalb des Prozessmodells. Wie in Unterabschnitt 2.1.5 erläutert, modellieren atomare Aktivitäten des WS-BPEL-Typs `Invoke`, `Receive` und `Reply` solche Aktivitäten, die mit prozessexternen WebServices über Operationsaufrufe und Parameterübergaben kommunizieren. Eine `AtomicActivity` kann im Gegensatz zu einer `ComplexActivity` lesend oder schreibend auf ein `Datum` zugreifen. Eine Aktivität greift lesend auf ein `Datum` zu, wenn das Datum als Eingabeparameter von der Aktivität verwendet wird; sie greift schreibend auf ein Datum zu, wenn das Datum als Ausgabeparameter verwendet wird. Daten können ebenfalls komplex oder atomar sein. Atomare Daten haben einen elementaren Datentyp. Komplexe Daten setzen sich aus anderen Daten zusammen. Sie bilden so Datenverbunde.

SimBPEL-Instance-Metamodell Prozessinstanzmodelle gleichen Prozessdefinitionsmodellen, tragen jedoch zusätzliche Informationen, wie beispielsweise Aktivitätszustände, die spezifisch für einen bestimmten Prozess zu einem bestimmten Zeitpunkt sind. Die Modellierungssprache für Prozessinstanzmodelle, namentlich SimBPEL-Instance, ist somit lediglich eine Erweiterung der Sprache für Prozessdefinitionsmodelle. Diesem Umstand wird in den Metamodellen insoweit Rechnung getragen, als dass die Metaklassen des unteren Drittels in Abbildung 2.14 die Metaklassen aus dem SimBPEL-

Metamodell um Metaklassen für besagte Informationen ergänzen. Somit stellt das untere Metamodell selbst kein eigenständiges Metamodell für Prozessinstanzmodelle dar. Das Metamodell für Prozessinstanzmodelle ist tatsächlich das SimBPEL-Metamodell zuzüglich der Metaklassen aus dem unteren Teil. Alternativ hätten die Metaklassen aus dem SimBPEL-Metamodell wiederholt modelliert werden können. Aus Gründen der Redundanzvermeidung ist die gewählte Metamodellierung jedoch vorzuziehen.

Aktivitäten werden in Prozessinstanzmodellen `ActivityInstances` zugeordnet. Letztere sind der Träger des Zustands der jeweiligen Aktivität zu einem bestimmten Zeitpunkt im Prozess. Der aktuelle Zustand wird dabei in einem Attribut `state` vorgehalten. Erlaubte Werte für die Zustände sind im Enumerationstypen `ActivityState` definiert.

Eine `Activity` kann laut Metamodell mehrere `ActivityInstances` referenzieren. Diese Sprachentwurfsentscheidung wurde gewählt, weil Aktivitäten gemäß der Kontrollflussdefinition tatsächlich mehrfach durchlaufen werden können, wenn sie Kinder von `While`-Aktivitäten sind. Aus der Anzahl der einer `Activity` zugeordneten `ActivityInstances` ist im Prozessinstanzmodell festgehalten, wie häufig eine Aktivität bereits durchlaufen wurde.

In den späteren Kapiteln wird der hier erläuterte Zusammenhang zwischen Prozessdefinitions- und -instanzmodellen aufgegriffen. Wie beschrieben sind Prozessinstanzmodelle prinzipiell Prozessdefinitionsmodelle, die um Prozesszustandsinformationen angereichert wurden. Umgekehrt betrachtet ist ein Prozessdefinitionsmodell identisch mit einem Prozessinstanzmodell, das einen Prozess im initialen Zustand modelliert. Aus einem beliebigen Prozessinstanzmodell lässt sich ein Prozessdefinitionsmodell gewinnen, indem alle zur SimBPEL-Instance-Metamodellerweiterung gehörigen Modellelemente entfernt werden, also alle `ActivityInstance`-Modellelemente.

2.3.2 Konkrete Syntax

Wie in Unterabschnitt 2.1.2 beschrieben, ist es günstig, bei einer Sprachdefinition den Sprachaufbau von der Repräsentation der Sprachzeichen zu trennen. Dies trifft insbesondere auf diagrammatische Sprachen zu. Im vorhergehenden Unterabschnitt wurde die abstrakte Syntax, also der Aufbau der Sprachausdrücke der verschiedenen Prozessmodellierungssprachen erläutert. Aus den Beispielen in Abschnitt 2.2 ist ersichtlich, dass die verschiedenen Modellelemente unterschiedliche visuelle Repräsentationen haben.

Die Zuordnungen zwischen abstrakter Syntax (Abbildung 2.14) und konkreter Syntax der jeweiligen Modellelemente geht aus den Abbildungen 2.10 und 2.11 hervor. Die Beschriftungen in den Klammern innerhalb dieser Abbildungen sind gerade die Namen der Metaklassen, von denen das jeweilige

Modellelement eine Modellinstanz ist.

Die Wurzeln der Kompositionshierarchien der verschiedenen Prozessmodelle haben keine visuelle Repräsentation. In allen drei Fällen ist es so, dass hier die Zeichenfläche die Modellelementinstanz des betreffenden Wurzelelements darstellt. Beispielsweise ist die Zeichenfläche von Abbildung 2.11 als Modellelementinstanz von `BpclModel` zu verstehen. Das Vorhandensein eine Modellelementinstanz auf der Zeichenfläche bedeutet gleichzeitig, dass das Element Teil der Instanz der jeweiligen `BpclModel`-Instanz ist, d.h. zwischen beiden besteht eine `activityTypes`-Beziehung (vgl. Abbildung 2.14). In SimBPEL- und SimBPEL-Instance-Modellen ist das visuelle Enthaltensein einer Aktivität innerhalb einer anderen Aktivität gleichbedeutend mit einer durch `childActivities` hergestellten Teil-Aggregat-Beziehung.

2.4 Formalismen zur Semantikdefinition

Die Semantik einer Modellierungssprache kann durch eine Abbildung in einem mathematischen Formalismus erklärt werden kann. Im Folgenden werden die für den Bereich der Prozessmodellierung gebräuchlichsten Formalismen eingeführt und dargelegt, weshalb für die Semantikdefinition von Prozessinstanz- und -definitionsmodell u.A. der Formalismus der Graphgrammatiken verwendet wurde.

2.4.1 Petri-Netze

Die Speicherplatzkomplexität der Berechnung allgemeiner rekursive Funktionen kann nicht a-priori ermittelt werden. Dies legt Rechnerarchitekturen nahe, deren Speicherressourcen bei Bedarf, d.h. während einer Berechnung, erweitert werden können. In seiner Dissertation [Pet62] beschreibt Carl-Adam Petri, dass die Kommunikationswege und somit auch die Signallaufzeit in solchen Architekturen mit fortschreitender Erweiterung zwangsweise ebenfalls steigen. Die mögliche Taktfrequenz eines zentralen Taktgebers sinkt dementsprechend ebenfalls. Petri schlug als Abhilfe eine Rechnerarchitektur mit autonomen, asynchron laufenden Komponenten vor. Sein Petri-Netz-Formalismus dient der Beschreibung solcher Architekturen.

Petri-Netz-Formalismus

Wie bei vielen mathematischen Formalismen gibt es keine einheitliche Definition für Petri-Netze. Eine einfache, hier verwendete Definition von Petri-Netzen ist [Aal97, Abschnitt 2] entnommen.

Definition 2.1 (Petri-Netz)
Ein Petri-Netz ist ein Tripel (P, T, F), worin

- P eine endliche Menge von Stellen (places),
- T eine endliche Menge von Transitionen (transitions) und
- $F \subseteq (P \times T) \cup (T \times P)$ eine Menge von Kanten ist (flow relation).

Betrachtet man P und T als Partitionen einer Knotenmenge, so ist ein Petri-Netz durch die Kantenrelation F ein bipartiter Graph. Eine Markierung $M \in P \rightarrow \mathbb{N}$ weist jeder Stelle eine Anzahl Tokens zu und definiert so einen bestimmten Zustand im Petri-Netz. Zu einem Zeitpunkt kann eine Transition feuern t, d.h. aus allen benachbarten Stellen, für die eine Kante zu t existiert, ein Token abziehen und in alle benachbarte Stellen, zu denen eine Kante von t existiert, ein Token platzieren. Hierdurch wird wiederum eine Struktur von Markierungen induziert, die ineinander übergehen können.

Varianten von Petri-Netzen unterscheiden sich vor allem in der Definition der Markierungsfunktion [BC92]. In manchen Definitionen ist M nur zweiwertig, also pro Stelle immer nur kein oder genau ein Token erlaubt, in anderen sind Tokens “gefärbt”, d.h. verschiedenen Klassen (Farben) zugeordnet und hierüber unterscheidbar. Bei letzteren Varianten spricht man auch von “high-level Petri-Nets”.

Petri-Netz-Modellierungssprachen

Der recht einfache Formalismus der Petri-Netze ist Grundlage für die Semantikdefinition einer Fülle von Modellierungssprachen, insbesondere im Bereich der Prozessmodelle. Bei diesen Sprachen kann man zwischen solchen unterscheiden, deren Semantik a-priori beim Sprachentwurf über Petri-Netze erklärt wurde und solchen, deren Semantik erst a-posteriori über Petri-Netze formal definiert wurde.

A-priori-Petri-Netz-Semantik Der direkteste, älteste und verbreiteteste Weg ist, den bipartiten Graph, den ein Petri-Netz darstellt, in üblicher Weise diagrammatisch zu notieren, wobei man für die Stellen Kreise verwendet, für die Transitionen Rechtecke und die Kanten als Pfeile zwischen Kreisen und Rechtecken notiert.

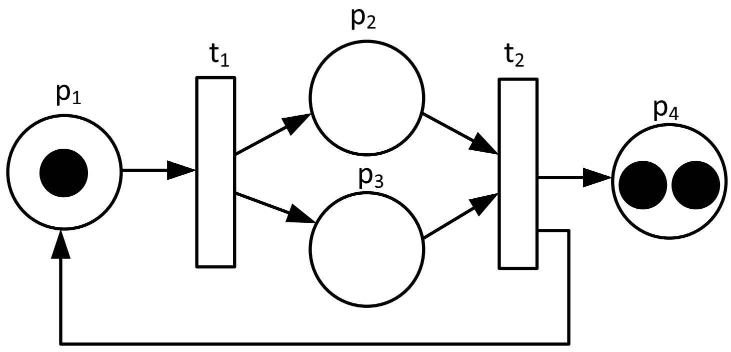


Abbildung 2.15: Beispiel eines Petri-Netzes

Beispiel 2.4 (Petri-Netz) Ein beispielhaftes Petri-Netz ist in Abbildung 2.15 abgebildet. Dieses Diagramm steht für ein markiertes Petri-Netz mit $P = \{p_1, p_2, p_3, p_4\}$, $T = \{t_1, t_2\}$ und $M(p_1) = 1$, $M(p_2) = M(p_3) = 0$, $M(p_4) = 2$.

In [Aal98, Abschnitt 4.1] definiert van der Aalst eine Unterklasse von Petri-Netzen, genannt *WF-Nets* (Workflow-Nets). WF-Nets sind Petri-Netze, die zwei ausgezeichnete Stellen haben (Ein- und Ausgabestelle) und die stark zusammenhängend sind. In [AH05] wird eine Petri-Netz-Variante EWF-Nets (Extended Workflow Nets) definiert, die ebenfalls die WF-Nets-Eigenschaften aufweist. Während mit Definition 2.1 beispielsweise nur nicht-deterministische Verzweigungen über eine Stelle mit zwei ausgehenden Kanten modellierbar sind, können in EWF-Nets auch deterministische Verzweigungen ausgedrückt werden. Zur Modellierung von deterministischen Verzweigungen wird die herkömmliche Petri-Netz-Definition um Steuerungsvariablen erweitert, die in prädikatenlogischen Ausdrücken evaluiert werden. Diese Ausdrücke sind den ausgehenden Kanten von speziellen Transitionen (explicit OR-splits) zugeordnet, die beim Feuern nur in den Stellen Token platzieren, die über eine Kante mit der Transition verbunden sind, deren Ausdruck unter der aktuellen Belegung der Steuerungsvariablen wahr ist. In [AH05] beschreiben Aalst et al. wie die Semantik der diagrammatischen Prozessmodellierungssprache YAWL (Abkürzung für “Yet Another Workflow Language”) über eine hierarchisch geordneten Menge von EWF-Nets dargelegt wird. Die Syntax von YAWL orientiert sich stark an der üblichen für Petri-Netze, erweitert um eigene Symbole für die hinzugekommenen Konzepte der EWF-Nets wie die “explicit OR-splits”.

A-posteriori-Petri-Netz-Semantik Neben der formalen Grundlage für neu entwickelte Prozessmodellierungssprachen werden Petri-Netze auch eingesetzt, um die Semantik bestehender Sprachen im Nachhinein zu definieren. In [MA07, MA06, Men07] geschieht dies für Ereignisgesteuerte Prozessketten (EPKs) [KNS92]. Die Business Process Modeling Notation (BPMN) wird in [DDO07] mittels Petri-Netzen formal erklärt, ebenso wie die in Unterabschnitt 2.1.5 erläuterte WebService Business Process Execution Language in [VA05, OVA+07, DDO07].

2.4.2 Graphgrammatiken in GROOVE

Ein Nachteil des Petri-Netz-Formalismus ist, dass er sich nur bedingt zur Beschreibung dynamischer Prozesse eignet. Bei Petri-Netzen wird normalerweise eine bestimmte Struktur, also Stellen, Transition und deren verbindende Kanten, für einen bestimmten Prozesstyp festgehalten. Eine Markierung entspricht dann einem bestimmten Zustand eines Prozesses. Mögliche Markierungen, also Prozesszustände, liegen demnach im Vorfeld bereits fest. In Abschnitt 1.2 wurde bereits diskutiert, dass diese Voraussetzung häufig nicht erfüllt werden kann.

Graphgrammatiken (synonym Graphersetzungssysteme) hingegen ermöglichen die Formalisierung von Prozesszuständen als Graphen und mögliche Zustandsänderungen als Graphersetzungsregeln. Zustandsübergänge können beliebige Graphänderungen herbeiführen. Insoweit können Graphersetzungsregeln nicht nur Zustandsänderungen herbeiführen, sondern auch dynamische Änderungen in der Ablaufstruktur des Prozesses, sofern diese im Graphen geeignet enthalten sind.

Es existiert eine Reihe von Werkzeugen, die der Erstellung von Graphgrammatiken dienen und die Ausführung ermöglichen, d.h. die tatsächliche (sukzessive) Anwendung der Ersetzungsregeln auf Wirtsgraphen. Hierzu gehören beispielsweise PROGRES [SWZ99] im Zusammenspiel mit der Laufzeitumgebung UPGRADE [BJSW02], AGG [Tae03], Fujaba (vgl. Vergleich in [FMRS07]), DiaGEN [PNB03] oder GROOVE [Ren03].

Unterschiede zwischen den Werkzeugen werden bei genauerer Betrachtung des zugrundeliegenden Formalismus deutlich. Es herrscht beispielsweise keine exakte Übereinstimmung darüber, welche formale Signatur ein Graph besitzt und auf welcher Basis Graphersetzungsregeln zu formalisieren sind. Diese Unterschiede rühren von der unterschiedlichen Zielsetzung der Werkzeuge her. PROGRES wurde beispielsweise mit der Maßgabe entwickelt, die Entwicklung interaktiver Werkzeuge für Entwicklungsprozesse zu unterstützen [Nag96]. Verknappt formuliert speichern solche Werkzeuge einen komplexen Entwicklungszustand, auch Konfiguration genannt, in einem Graphen, der vom Benutzer durch Anwendung von Graphersetzungsregeln sukzessive geändert werden kann.

GROOVE bezweckt demgegenüber Verifikationsunterstützung objektorientierter Systeme [Ren03]. Zustände dieser Systeme lassen sich ohne paradigmatischen Bruch als Graphen auffassen, wobei Knoten als Objekte und Kanten als Referenzen zwischen Objekten aufgefasst werden. Durch Graphersetzungsregeln können in GROOVE Änderungen in den Objektstrukturen spezifiziert werden, also die Erzeugung oder Löschung von Objekten, Objektattributwerten oder Referenzen.

In dieser Arbeit wird GROOVE an verschiedenen Stellen zu verschiedenen Zwecken eingesetzt:

- In Abschnitt 2.5 wird eine GROOVE-Grammatik zur Semantikdefinition von Prozessinstanzmodellen verwendet.
- In Kapitel 3 wird GROOVE eingesetzt, um bestimmte Transformationen von Prozessdefinitionsmodellen zu erläutern.
- In Kapitel 5 ist GROOVE Teil der Realisierung einer Ähnlichkeitsberechnung von Prozessinstanzmodellen.

Definition 2.2 (Graphgrammatik) Eine *Graphgrammatik* $\mathcal{G} = (\mathcal{R}, s)$ ist ein Paar bestehend aus

- einem Startgraph $s \in \mathsf{Graph}$ (vgl. Definition 2.3) und
- einer Menge $\mathcal{R}$ von *Graphersetzungsregeln* (vgl. Definition 2.4).

Die Regeln in $\mathcal{R}$ beziehen sich auf einen *Wirtsgraphen* (*Arbeitsgraphen*), der durch Anwendungen der Regeln manipuliert, d.h. in einen anderen Graphen überführt werden kann, beginnend mit dem Startgraphen s.

Graphen Das Anwendungsbeispiel ist eine Ausführungssemantik einer Sprache für einfache Prozessinstanzmodelle. Die abstrakte Syntax eines Prozessinstanzmodells kann als Graph aufgefasst werden. In GROOVE werden Graphen als Graphdiagramme notiert. Ein Beispiel hierfür ist der Graph s_1, der in Abbildung 2.16 als Graphdiagramm dargestellt ist. Verkürzt gesprochen ist das Graphdiagramm die abstrakte Syntax des Prozessinstanzmodells.

In den Prozessinstanzmodellen des Beispiels gibt es nur eine Art von Aktivitäten, genannt Activity. Aktivitäten können entweder im Zustand waiting, active oder finished sein. Zu Anfang sind immer alle Aktivitäten im Zustand waiting. Aktivitäten stehen in Kontrollfluss- (link) oder Verfeinerungsbeziehung (child) zueinander. Diese Beziehungen schränken mögliche Zustandswechsel in Aktivitäten folgendermaßen ein:

link Eine Aktivität kann nur von waiting in active wechseln, wenn alle ihre link-Vorgänger in *finished* sind. Dies ist trivialerweise insbesondere dann erfüllt, wenn einen Aktivität keine link-Vorgänger hat.

child Eine Aktivität kann nur von active in finished wechseln, wenn alle ihre child-Nachfolger (ihre Kinder) in finished sind. Das ist wiederum trivialerweise dann erfüllt, wenn eine Aktivität keine Kinder hat.

Definition 2.3 (Graph) Ein Graph ist nach [Ren08, Definition 1] ein Tupel (V, E) mit

- $V \subseteq \mathsf{Node}$ einer Menge von Knoten und
- $E \subseteq V \times \mathsf{Label} \times V$ einer Menge von gerichteten, beschrifteten Kanten.

In den Graphdiagrammen werden Knoten als Rechtecke notiert und Kanten als Pfeile. In GROOVE können nur Kanten beschriftet werden. Beschriftungen (ohne Gleichheitszeichen) in den Rechtecken sind nur die konkrete Syntax für reflexive Kanten an dem betreffenden Knoten, also Kanten, die den betreffenden Knoten sowohl als Quelle als auch als Ziel haben. Der oberste Knoten des Graphen s_1 hat also die reflexive Kante Activity, sowie zwei child-Kanten zu anderen Knoten. Diese reflexiven, beschrifteten Kanten werden im Weiteren vereinfachend Knotenbeschriftungen genannt. Eine weitere Verdichtung stellen Beschriftungen mit Gleichheitszeichen in den Rechtecken wie name = “A” dar. Diese Beschriftung repräsentiert eine Kante name zu einem sogenannten Datenknoten, der selbst die Knotenbeschriftung string:”Main” trägt. Über Datenknoten können Attribute von Knoten modelliert werden, die beispielsweise als Zeichenkette (string) oder Ganzzahl (int) typisiert sind.

Definition 2.4 (Graphersetzungsregel) Eine Grapherserzungsregel $g = (l, r) \in \mathcal{R}$ ist ein Paar aus zwei Graphen l und r, genannt linke und rechte Regelseite. Eine Graphersetzungsregel g ist anwendbar auf einen Graphen s, wenn l zu s passt. Die dann mögliche Anwendung von g auf s überführt s in einen neuen Graphen s', in dem die Anwendungsstelle von l,

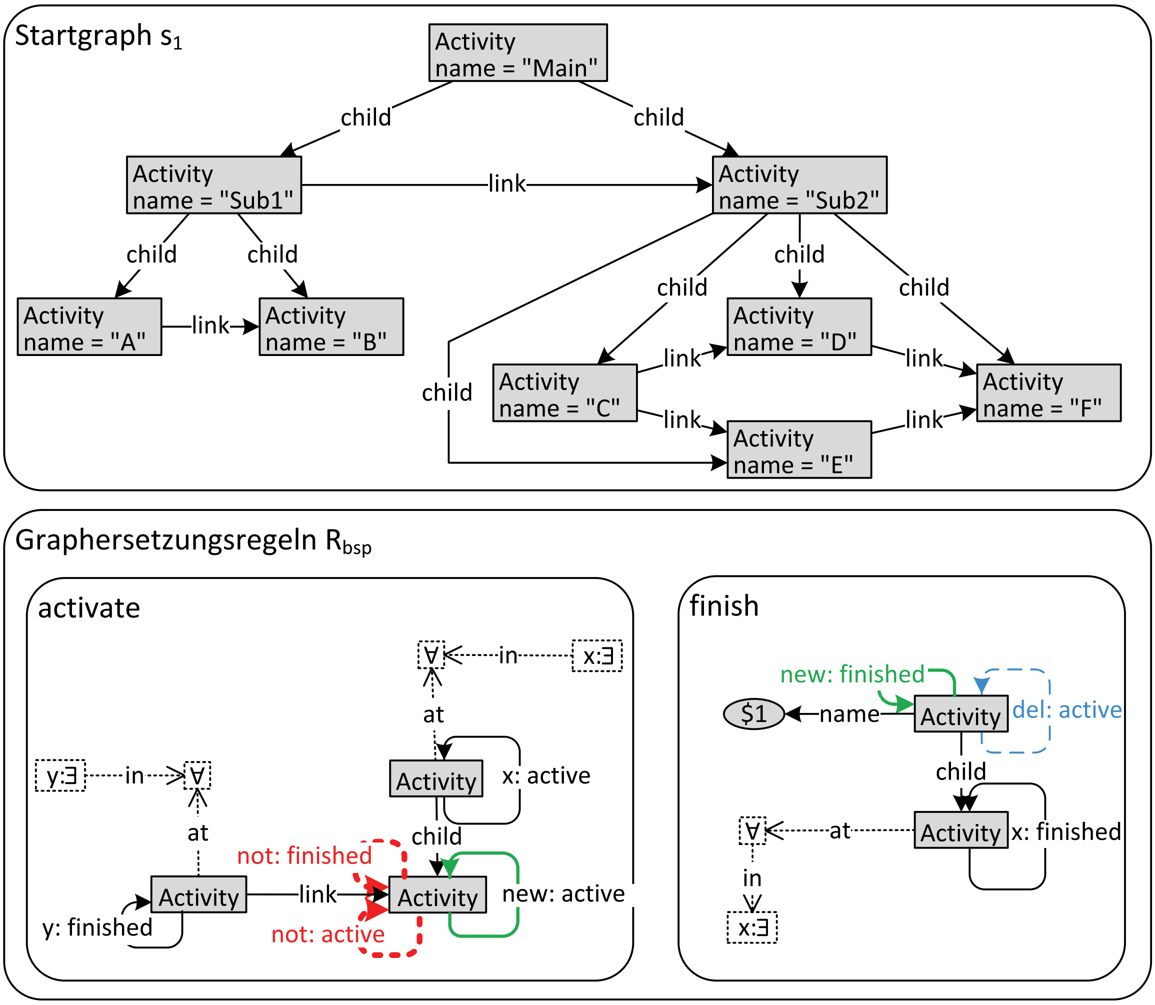


Abbildung 2.16: Beispielgrammatik $\mathcal{G}_{\text{bsp}} = (\mathcal{R}_{bsp}, s_1)$

ein Teilgraph von s, durch r ersetzt wird. Eine formale Erläuterung der Regelanwendbarkeit und Überführung findet sich in [Ren08, Abschnitt 2].

Graphersetzungsregeln Graphersetzungsregeln werden in GROOVE ebenfalls als Graphdiagramme notiert. In diesen Graphdiagrammen ist die linke und rechte Graphersetzungsregelseite zusammengefasst. Die Zuordnung zu einer Regelseite geschieht durch folgende Auszeichnungen der Knoten und Kanten durch spezielle Beschriftungspräfixe:

use Mit use beschriftete Knoten und Kanten werden der linken Regelseite zugeordnet. Passt die Regel auf einen Teilgraphen eines Graphen s, so ist insbesondere das durch use-Knoten und -Kanten gebildete Muster vorhanden. Use-Elemente werden in der konkreten Graphdiagrammsyntax durch eine dünne, durchgezogene, schwarze (umrandende) Linie dargestellt.

not Mit not können bestimmte Muster ausgeschlossen werden. Hiermit ausgezeichnete Knoten/Kanten gehören zur linken Regelseite. Sie sind nicht Bestandteil eines Teilgraphen, auf den die Regel anwendbar ist. Not-Elemente werden durch dicke, strichlierte, rote (umrandende) Linien dargestellt.

new Mit new ausgezeichnete Knoten/Kanten werden durch die Regelanwendung erzeugt. Sie sind daher Bestandteil der rechten Regelseite. Dargestellt werden sie durch dicke, durchgezogene, grüne (umrandende) Linien.

del Mit del ausgezeichnete Knoten/Kanten werden durch die Regelanwendung gelöscht. Sie sind daher auch Bestandteil der rechten Regelseite. Dargestellt werden sie durch dünne, strichlierte, blaue (umrandende) Linien.

Ist ein Knoten oder eine Kante mit keinem Präfix beschriftet, so ist per Konvention die Beschriftung implizit use.

Elemente in Graphersetzungregeln können quantifiziert werden. Dies wird durch spezielle, mit $\forall$ bzw. $\exists$ beschriftete Knoten in den Graphersetzungsregeln ausgedrückt. Durch diese Quantoren kann über Graphmuster quantifiziert werden. Die Auswirkung dieser Quantoren wird hier nur beispielhaft erläutert; eine formale Semantikdefinition findet sich in [RK09].

Die Regelmenge des Beispiels besteht aus zwei Ersetzungsregeln: $\mathcal{R}_{bsp} = (\text{activate}, \text{finish})$. Sie ändern Aktivitätszustände, wobei diese in den Graphen Knotenbeschriftungen active und finished sind. Eine Aktivität ist im Zustand waiting, wenn sie keine dieser Beschriftungen aufweist.

activate Zentral in der activate-Regel ist der Activity-Knoten unten rechts. Passt die Regel, wird eine reflexive active-Kante erzeugt. Voraussetzung hierfür ist allerdings,

- dass der Knoten keine reflexive active- oder finished-Kante aufweist,
- dass *für alle* ($\forall$) link-Vorgänger gilt, dass sie jeweils ($\exists$) eine reflexive finished-Kante haben und
- dass für alle ($\forall$) child-Vorgänger (Eltern in der Verfeinerungsbeziehung) gilt, dass sie jeweils ($\exists$) eine reflexive active-Kante aufweisen.

finish Zentral in der finish-Regel ist der Activity-Knoten oben. Passt diese Regel, so wird die entsprechende, reflexive active-Kante gelöscht und eine reflexive finish-Kante erzeugt. Voraussetzung hierfür ist, dass alle ($\forall$) Activity-Kinder jeweils eine reflexive finished-Kante haben. Der mit $1 beschriftete Knoten passt auf den name-Datenknoten des Activity-Knotens. Die Bedeutung seiner besonderen Beschriftung wird später erläutert.

Beispiel 2.5 (Regelanwendung) Auf den Graphen s_1 oben in Abbildung 2.16 passt die activate-Regel genau einmal, die finish-Regel keinmal. In der activate-Regel passt der linke untere Knoten auf den obersten Knoten des Graphen. Dieser hat weder Eltern noch Vorgänger, weswegen die quantifizierten Muster der Regel trivialerweise erfüllt sind und hat auch (noch) keine reflexiven active- oder finished Kanten, weswegen die Regelanwendung auch nicht durch die not-Kanten verhindert wird. Dass der Knoten zusätzlich mit Main beschriftet ist, hat keine Auswirkungen auf die Regelanwendbarkeit, was auch für die zusätzlichen Beschriftungen der anderen Activity-Knoten gilt.

Graphtransitionssystem Eine Regelanwendung überführt einen Graphen in einen anderen. Fasst man Graphen als Zustände auf und Regelanwendungen als Zustandsübergänge, so induziert eine Graphgrammatik $\mathcal{G} = (\mathcal{R}, s)$ mit Startgraph s und Regelmenge $\mathcal{R}$ ein Graphtransitionssystem.

Definition 2.5 (Graphtransitionssystem) Ein Graphtransitionssystem $\mathfrak{T}_{\mathcal{G}}$ einer Graphgrammatik $\mathcal{G}$ ist ein $\{R, L\}$-Struktur über dem Träger S_p mit

- $S_p \subseteq \mathsf{Graph}$, der Menge der Zustände (Graphen),
- $R \subseteq S_p \times AP \times S_p$, der Zustandsübergangsrelation, also einer Menge von beschrifteten Zustandsübergängen, und
- $L : S_p \to 2^{AP}$, eine Funktion, die einem Zustand eine Menge AP von atomaren Aussagen zuordnet, die in dem Zustand gelten.

Ein Zustandsübergang, also ein Element $(s, a, s') \in R$ wird der Lesbarkeit halber mit $s \xrightarrow{\text{a}} s'$ notiert. Per Konvention gelten in einem Zustand s insbesondere die atomaren Aussagen seiner ausgehenden Transitionen, also

$$s \xrightarrow{\text{a}} s' \in R \Rightarrow a \in L(s).$$

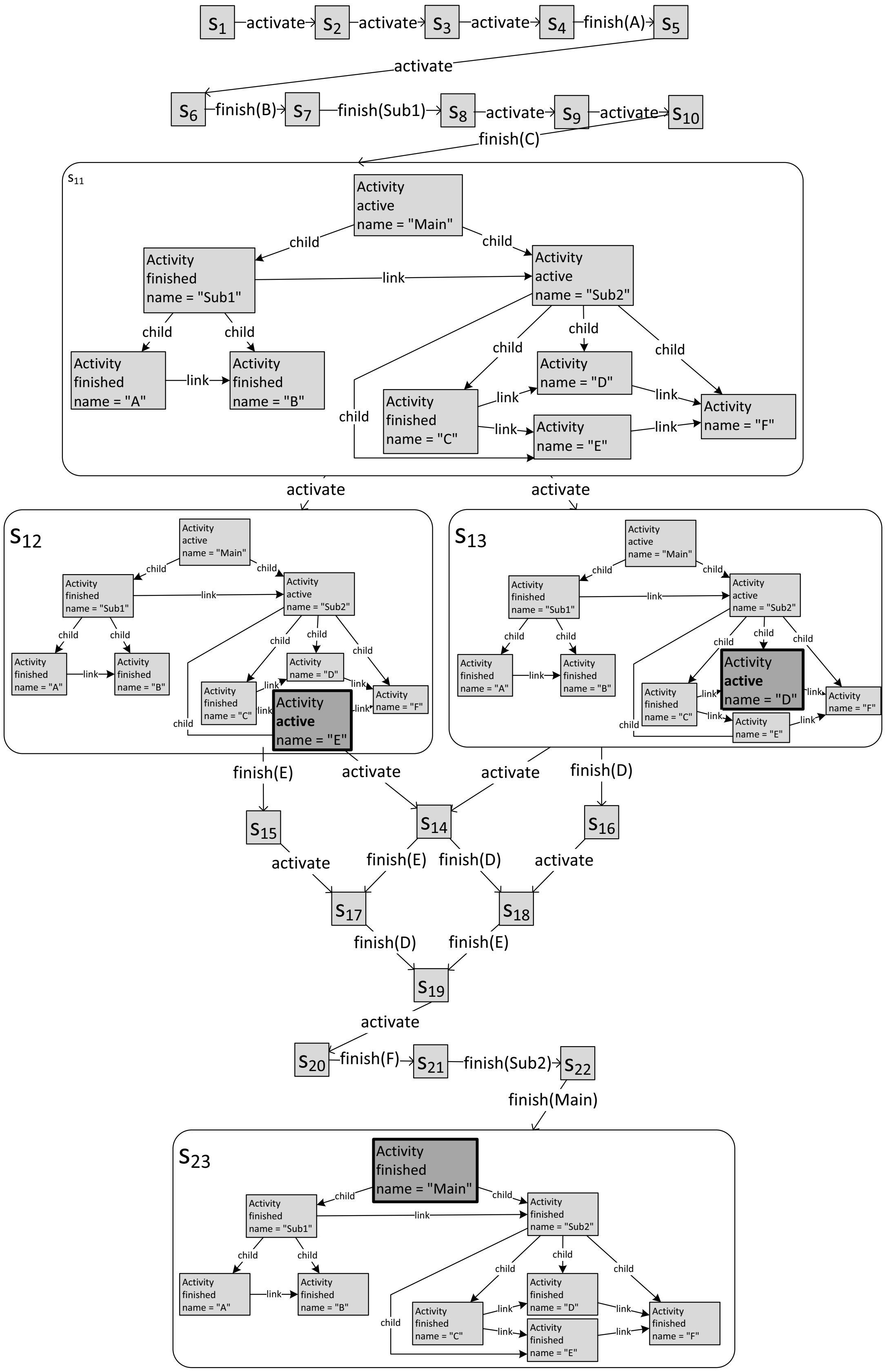


Abbildung 2.17: Graphtransitionssystem $\mathfrak{T}_{\mathcal{G}_{\text{bsp}}}$ für Graphgrammatik $\mathcal{G}_{\text{bsp}}$ aus Abbildung 2.16

Beispiel 2.6 (Transitionssystem $\mathfrak{T}_{\mathcal{G}_{\mathrm{bsp}}}$) Abbildung 2.17 zeigt das von Abbildung 2.16 induziert Graphtransitionssystem $\mathfrak{T}_{\mathcal{G}_{\mathrm{bsp}}}$, wobei die Zustandsgraphen aus Platzgründen nur von einigen ausgewählten Zuständen gezeigt sind. Die Übergänge zwischen den Zuständen sind jeweils mit dem Namen der Regel beschriftet, deren Anwendung den Zustandswechsel verursacht. Zu beachten ist, dass von einem Zustand s ausgehend potentiell auch mehrere Zustände s'_i erreicht werden können. Dies ist der Fall, wenn mehrere verschiedene Regeln auf den betreffenden Graph s passen oder eine Regel mehrere Anwendungsstellen, also mehrere passende Teilgraphen in einem Graphzustand hat oder beides zutrifft. Im Beispiel von Abbildung 2.17 kann im Zustand s_{11} entweder dem Activity-Knoten D eine active-Knotenbeschriftung hinzugefügt werden, was zum Zustand s_{13} führt oder dem Activity-Knoten E, was zum Zustand s_{12} führt.

Zuständen in Transitionssystemen werden atomare Aussagen (engl.: atomic propositions) AP zugeordnet. In einem Zustand kann dabei eine atomare Aussage gelten oder nicht. Bei Graphtransitionssystemen sind die atomare Aussagen Graphersetzungsregeln. Eine Graphersetzungsregel gilt in einem Zustand, wenn Sie anwendbar ist. Dabei ist es unerheblich, ob die Anwendung der Regel einen Zustandswechsel herbeiführt oder nicht. Im Transitionssystem $\mathfrak{T}_{\mathcal{G}_{\mathrm{bsp}}}$ gilt im Startzustand s_1 (vgl. Abbildung 2.16) beispielsweise nur die atomare Aussage activate, im Zustand s_{11} finish. Neben den Regelnamen können in Graphtransitionssystemen atomare Aussagen auch Parameter aufweisen. Im Übergang von s_{17} nach s_{19} lautet dieser beispielsweise D, die atomare Aussage also insgesamt finish(D). Der Aktualwert des Parameters wird durch den Datenknoten bestimmt, auf den der mit \$1 beschriftete Knoten in der finish-Regel aus Abbildung 2.16 passt.

Beispiel 2.7 (Atomare Aussagen) In $\mathfrak{T}_{\mathcal{G}_{\mathrm{bsp}}}$ ist die Menge atomarer Aussagen

$$AP = \{\mathsf{activate()},\mathsf{finish(a)},\ldots,\mathsf{finish(Sub1)},\mathsf{finish(Main)}\}.$$

Ein Graphtransitionssystem kann als (potentiell unendliche) Menge von beliebig langen Pfaden aufgefasst werden. Ein Pfad ist dabei ein Wort über dem Alphabet S_p der Zustände des Graphtransitionssystems.

Definition 2.6 (i-tes Symbol) In einem Wort w über einem beliebigen Alphabet steht w_i für das Symbol an i-ter Stelle. Das erste Symbol ist per Konvention w_0.

Definition 2.7 (k-Suffix) Für ein Wort $w = w_0 w_1 \ldots$ wird das Teilwort $w_k w_{k+1} w_{k+2} \ldots$ als k-Suffix von w bezeichnet und der Kürze halber als $w_{k\ldots}$ notiert.

Definition 2.8 (Wortlänge) Sei $w = w_1 \ldots w_n$ mit $n \in \mathbb{N}_0$ ein Wort. Dann ist $|w| := n$ die Länge des Worts, d.h. die Anzahl seiner Symbole.

Definition 2.9 (Vorkommenshäufigkeit) Sei $w = w_1 \ldots w_n$ mit $n \in \mathbb{N}_0$ ein Wort über dem Alphabet A und $a \in A$ ein Symbol aus dem Alphabet. Dann ist

$$|w|_a := |\{i \in \{1, \ldots, n\} \mid w_i = a\}|$$

die Vorkommenshäufigkeit von a in w.

Definition 2.10 (S_p-Sprache) Die zu einem Graphtransitionssystem $\mathfrak{T}_{\mathcal{G}}$ mit der Zustandsmenge S_p und der Transitionsrelation R gehörige S_p-Sprache $\mathcal{L}_{S_p}(\mathfrak{T}_{\mathcal{G}})$ ist definiert als

$$\mathcal{L}_{S_p}(\mathfrak{T}_{\mathcal{G}}) := \{\pi \in S_p^* \mid \forall i \in \mathbb{N}_0\ \exists a \in AP : (\pi_i, a, \pi_{i+1}) \in R\}.$$

Benachbarte Symbole entsprechen also benachbarten Zuständen gemäß der Transitionsrelation R.

Beispiel 2.8 (S_p-Wort) Folgendes Wort gehört zur S_p-Sprache des Graphtransitionssystems $\mathfrak{T}_{\mathcal{G}_{\text{bsp}}}$:

$$\pi = s_1 s_2 \dots s_{10} s_{11} s_{12} s_{14} s_{18} s_{19} s_{20} \dots s_{23} \in \mathcal{L}_{S_p}(\mathfrak{T}_{\mathcal{G}_{\text{bsp}}})$$

Zwar bilden Pfade in Graphtransitionssystemen bereits Wörter, allerdings ist das zugrundeliegende Alphabet S_p ungeeignet für weitere Formalisierungen, da die darin enthaltenen Symbole Graphen sind. Die folgende Definition vermittelt zwischen den komplexen Symbolen aus S_p und einfachen aus dem Alphabet $AP' \subseteq AP$.

Definition 2.11 (AP'-Sprache) Die zu einem Graphtransitionssystem $\mathfrak{T}_{\mathcal{G}}$ gehörige AP'-Sprache mit $\mathcal{L}_{AP'}(\mathfrak{T}_{\mathcal{G}})$, wobei $AP' \subseteq AP$, ist wie folgt definiert:

$$\begin{aligned}
&\mathcal{L}_{AP'}(\mathfrak{T}_{\mathcal{G}}) := \{w \in AP'^{*} \mid \exists \pi \in \mathcal{L}_{S_p}(\mathfrak{T}_{\mathcal{G}}) \wedge \exists f : \mathbb{N}_0 \to \mathbb{N}_0 \ : \\
&\forall i \in \{0, \dots, |w| - 1\} : \ f(i) < f(i+1) \wedge w_i \in L(\pi_{f(i)}) \\
&\ \wedge \forall k \in \{f(i) + 1, \dots, f(i+1) - 1\} : L(\pi_k) \cap AP' = \emptyset\}
\end{aligned}$$

Im Kern besagt die Definition, dass ein Wort w in $\mathcal{L}_{AP'}(\mathfrak{T}_{\mathcal{G}})$ ist, wenn ein Pfad durch $\mathfrak{T}_{\mathcal{G}}$ existiert, auf dem atomare Aussagen aus AP' in der durch w gegebenen Reihenfolge gelten. Zwischenzustände, auf denen keine atomaren Aussagen aus AP' gelten, fallen aus der Betrachtung heraus.

Beispiel 2.9 (AP'-Wort) Folgendes Wort w ist aus der AP'-Sprache, die zum Graphtransitionssystem $\mathfrak{T}_{\mathcal{G}_{\text{bsp}}}$ gehört. Dabei ist das Alphabet $AP' = \{\mathsf{finish(A)}, \dots, \mathsf{finish(F)}\}$.

$$w = \mathsf{finish(A), finish(B), finish(C), finish(D), finish(E), finish(F)}$$

Das Beispielwort ist gerade so gewählt, dass es mit dem S_p-Wort aus Beispiel 2.8 korreliert. Hierbei ist z.B. finish(C) eine atomare Aussage des Zustands s_{10}. Die Kommata zwischen den Symbolen dienen nur der besseren Lesbarkeit, haben aber sonst keine weitere Bedeutung.

2.4.3 Eignung für dynamische Prozesse

Die Semantik von Prozessmodellierungssprachen wird normalerweise operational definiert, da beim Prozessmanagement die Wechselwirkung zwischen Prozesszustand (Prozessinstanz) und der Realwelt im Vordergrund steht und nicht beispielsweise eine durch das Prozessmodell berechnete Funktion. Je nach Art des Formalismus und damit je nach Sprache liegt dabei ein unterschiedliches Zustandsmodell zugrunde.

Die Abbildung 2.18 verdeutlicht die Unterschiede. Sie zeigt, in graphischer Notation, das Petri-Netz aus Beispiel 2.4. Bei Petri-Netzen ist ein Zustand s_i gleichzusetzen mit einer bestimmten Markierung M_i, die sich laut Definition 2.1 auf eine bestimmte Petri-Netz-Struktur bezieht. Zustandswechsel können automatisch geschehen oder durch externe Ereignisse angeregt werden.

Ähnliches gilt auch für imperative Programmiersprachen. Hier ist der Zustandsraum, vereinfacht darstellt, eine Menge von Variablenbelegungen (beispielsweise $x \leftarrow 5, y \leftarrow 7, z \leftarrow 0$ im Zustand s_0) und die Position des Programmzählers (beispielsweise l_0 im Zustand s_0). Gemein ist diesen Ansätzen, dass zwar der Zustand eines Prozesses dynamisch ist, jedoch die Prozess-/Programmstruktur im Zeitraum der Prozessausführung statisch bleibt. Die Zustände beziehen sich zwar auf die Festlegungen in der Definitionsebene, beinhalten aber selbst nicht die dort enthaltene Ablaufstruktur, die die möglichen Zustandswechsel einschränkt.

Auf Graphgrammatiken basierende Ansätze unterscheiden sich in diesem Punkt fundamental von den oben genannten Ansätzen. Da der Zustandsraum in diesem Fall Graphen sind, ist es hier möglich, die Kontrollflussabhängigkeiten selbst in den Zuständen als Kanten festzuhalten. Auf Definitionsebene finden sich Graphersetzungregeln, die zum einen Ausführungszustände unter Berücksichtigung der Kontrollflussabhängigkeiten ändern (Regel ③). Zum anderen aber können diese Regeln auch strukturelle Änderungen in diesen Abhängigkeiten bewirken. Regel ① erzeugt beispielsweise neue Aktivitäten; Regel ② verbindet bestehende Aktivitäten mit einer Kontrollflusskante, wobei zur Zyklenvermeidung kein entgegengesetzter Kontrollflusspfad existieren darf.

Da Ausführungszustände und Kontrollflussdefinitionen in Graphgrammatiken in einem einzigen komplexen Graph zusammenfallen und nicht getrennt voneinander gehalten werden, sind Graphgrammatiken im Umfeld von dynamischen Prozessmanagementsystemen nützlich und eignen sich sogar als Realisierungswerkzeug für vollständige Prozessmanagementsysteme, wie im Vorläuferprojekt AHEAD bewiesen wurde. Diese direkte Form des Einsatzes von Graphgrammatiken konnte in dieser Arbeit nicht weitergeführt werden, da hier auf basierenden Prozessmanagementwerkzeugen aufgesetzt wurde.

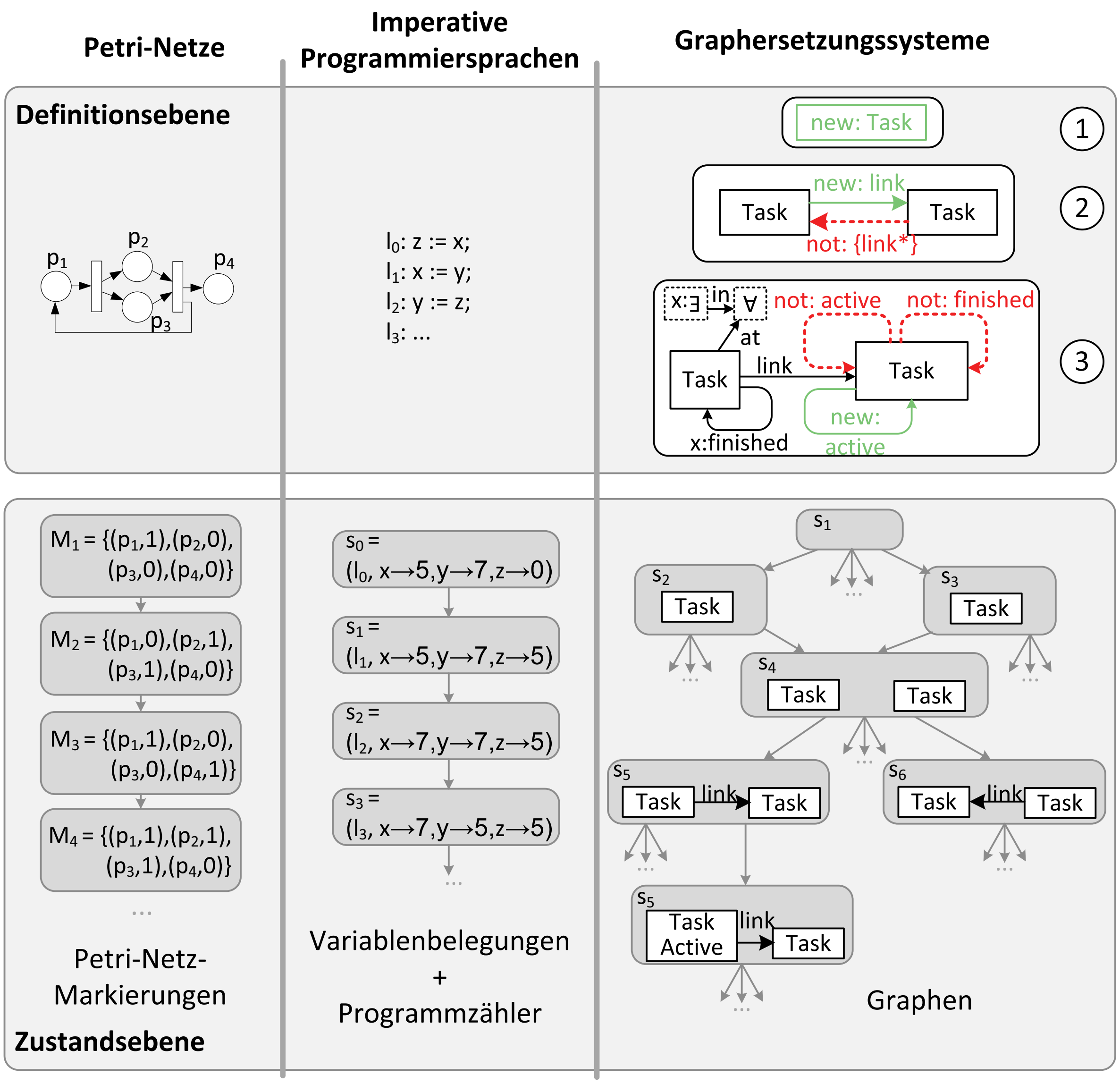


Abbildung 2.18: Vergleich von Transitionssystem-erzeugenden Formalismen

Trotzdem sind Graphgrammatiken auch in dieser Arbeit wichtig und zwar als Teil einer Konzeptrealisierung wie in Kapitel 5, zum Entwurf und Darstellung anderweitig realisierter Konzepte wie in Kapitel 3 und zur Unterstützung der formalen Semantikdefinition von Prozessinstanzmodellen wie im folgenden Abschnitt beschrieben.

2.5 Semantikdefinition für Prozessinstanzmodelle

Im vorhergehenden Abschnitt wurde die Syntax der Prozessmodelle formalisiert. Insbesondere wurde die Definition der abstrakten Syntax der Sprache

WS-BPEL von XML-Schema mit bestimmten Vereinfachungen und Erweiterungen nach Ecore übertragen. Die dadurch entstandene Sprache wird SimBPEL genannt. Durch Erweiterung von SimBPEL lässt sich auf einfache Weise eine Modellierungssprache für Prozessinstanzmodelle (SimBPEL-Instance) gewinnen.

Durch besagtes XML-Schema liegt also für WS-BPEL eine formale Definition zumindest der abstrakten Syntax vor. Auf die WS-BPEL-Semantik trifft das nicht zu; insoweit ist es also auch nicht möglich, die Semantikdefinition auf SimBPEL zu übertragen.

Eine formale Semantikdefinition ist jedoch unerlässlich, um weitere Ergebnisse dieser Arbeit präzise zu beschreiben. Im Folgenden wird daher die Semantik von Prozessinstanzmodellen in SimBPEL-Instance formal definiert.

Wie in Unterabschnitt 2.1.2 erläutert, ist die Semantikdefinition einer Modellierungssprache lediglich ein Abbildung ihrer Ausdrücke in eine andere Sprache oder Formalismus – die semantische Domäne [HR04] – für die bzw. den eine Semantik definiert ist. Prozessinstanzmodelle modellieren Sachverhalte verschiedener Aspekte. Somit wird die Semantik von Prozessinstanzmodellen nicht durch eine einzelne Abbildung erklärt, sondern durch eine pro Aspekt. Die Semantische Domäne von Prozessinstanzmodellen ist daher dreigeteilt in Wörter, Graphen und Graphtransitionssysteme.

Zustandsaspekt Aus einem Prozessinstanzmodell geht der *aktuelle Ausführungszustand* des modellierten Prozesses hervor, d.h. in einem Prozessinstanzmodell ist ablesbar, welche Aktivitäten sich in welchem Zustand befinden.

Historienaspekt Aus einem Prozessinstanzmodell geht neben dem aktuellen Zustand auch die *Prozesshistorie* aus den `preceded`-Referenzen hervor. Man kann einem Prozessinstanzmodell also entnehmen, welche abgeschlossenen Aktivitäten bisher in welcher Reihenfolge durchgeführt wurden.

Zukunftsaspekt Prozessinstanzmodelle sind syntaktisch betrachtet Erweiterungen von Prozessdefinitionsmodellen bzw. Prozessdefinitionsmodelle Spezialfälle von Prozessinstanzmodellen. Insbesondere ist in Prozessinstanzmodellen auch die Kontrollflussdefinition festgehalten. Es ist also möglich, anhand eines Prozessinstanzmodells, das einen bestimmten Prozesszustand modelliert, zu entscheiden, welche Reihenfolgen von Aktivitätsdurchführungen ausgehend von dem aktuellen Prozesszustand noch möglich sind.

In den folgenden Unterabschnitten wird je eine Semantikdefinition für einen der genannten Blickwinkel von Prozessinstanzmodellen erläutert. Die Definition der Semantik für den aktuellen Zustand (vgl. Unterabschnitt 2.5.1) und die Historie (vgl. Unterabschnitt 2.5.2) ist dabei einfach, die für die mögliche Zukunft, also die Ausführungssemantik dagegen komplex (vgl. Unterabschnitt 2.5.3).

2.5.1 Zustandssemantik

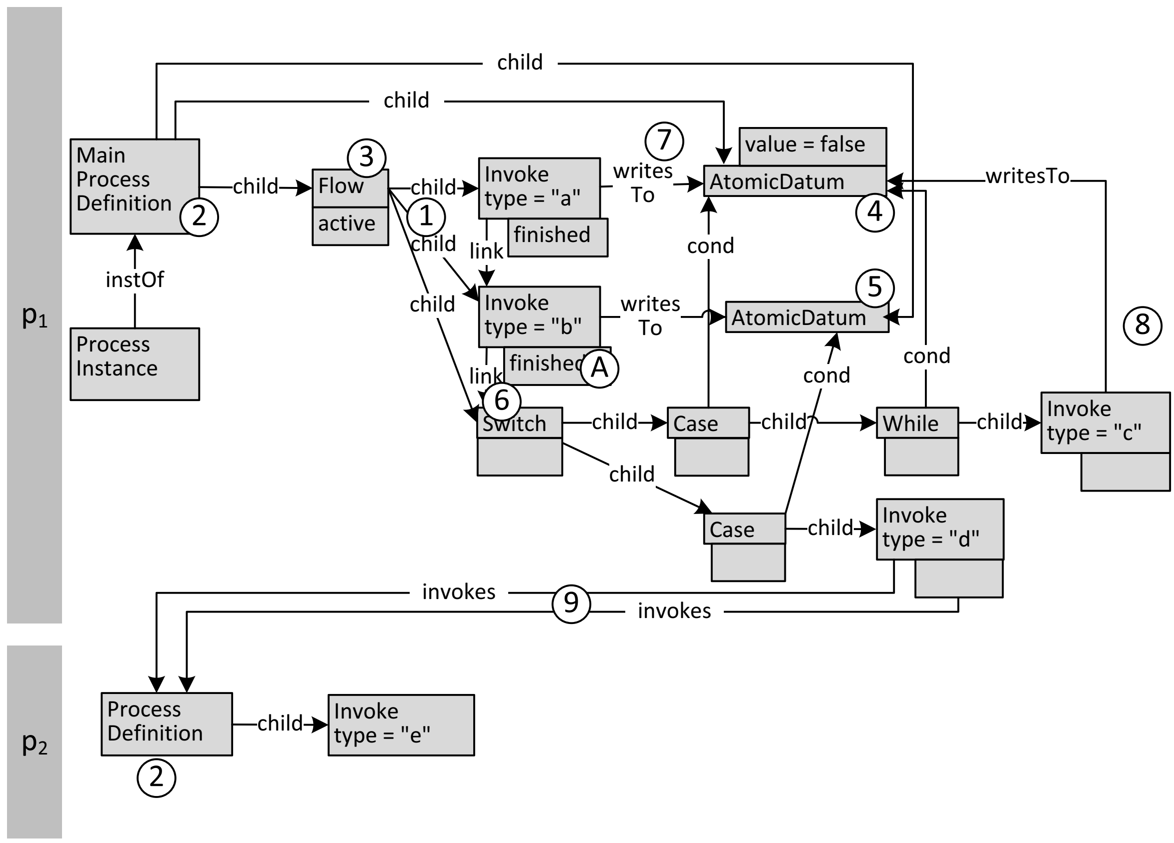
Prozessinstanzmodelle modellieren insbesondere Prozesszustände. Die Semantische Domäne der Prozesszustände ist in dieser Arbeit die Menge spezieller Graphen gemäß Definition 2.3. Diese werden im Folgenden als *Prozessgraphen* bezeichnet.

Beispiel 2.10 (Prozessgraph eines -instanzmodells) Sei M die Menge aller möglichen Instanzmodelle und Graph die Menge aller möglichen Graphen. Dann ist

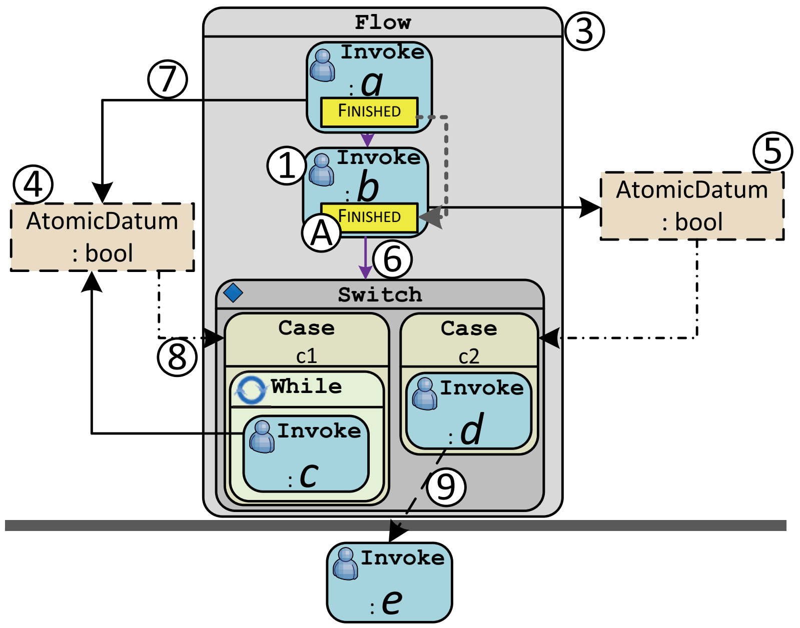
$$\mathrm{pg} : 2^M \rightarrow \mathsf{Graph}$$

eine Abbildung, die jeder Menge von Prozessinstanzmodellen einen Prozessgraphen zuordnet. Das Urbild ist immer eine Menge von Prozessinstanzmodellen, da Prozessinstanzmodelle untereinander in Aufrufbeziehung stehen können. Die Aufrufbeziehung muss sich im Bild wiederfinden. Im Folgenden wird mit $\mathrm{pg}(\{p\})$ (verkürzt $\mathrm{pg}(p)$) die *Zustandssemantik* eines Prozessinstanzmodells p bezeichnet. Die Abbildungsvorschrift ist trivial aber umfangreich. Sie ist daher in Abbildung 2.19 nur beispielhaft dargestellt, wobei aus der Nummerierung die Zuordnung hervorgeht.

Abbildung 2.19(a) zeigt einen beispielhaften Prozessgraphen. Er definiert die Semantik zweier Prozessinstanzmodelle p_1 und p_2. In der konkreten Syntax der Prozessinstanzmodelle ist die Kompositionshierarchie durch visuelles Enthaltensein widergespiegelt; im Prozessgraphen durch child-Kanten (z.B. ①). Die Wurzel der Kompositionshierarchien im Prozessgraphen bildet dabei Knoten, die eine Beschriftung Process besitzen ②. In der konkreten Syntax der Prozessinstanzmodelle ist das Process-Element nicht direkt dargestellt sondern entspricht der Zeichenfläche. Darauf befindet sich genau ein Kind, das die Wurzel der Kompositionshierarchie der Aktivitäten darstellt, im oberen Prozessinstanzmodell ein Flow ③. Geschwister des Flows sind AtomicDatum-Prozessdaten (④ und ⑤). Aktivitäten innerhalb des Flows sind durch link-Kanten verbunden (z.B. ⑥). Schreibende Zugriffe auf Prozessdaten (AtomicDatum) werden durch writesTo-Kanten etabliert (z.B. ⑦). Lesende Zugriffe finden in dem Beispiel nur durch die Case- und den While-Knoten statt (z.B. ⑧), die jeweils ein AtomicDatum in ihren Bedingungen referenzieren



(a) Prozessgraph $\mathrm{pg}(\{p_1, p_2\})$



(b) Prozessinstanzmodelle p_1 und p_2

Abbildung 2.19: Prozessinstanzmodelle mit semantikdefinierendem Prozessgraph

(Kante cond). Insbesondere schreibt die in der While-Aktivität enthaltene Invoke-Aktivität c in die Abbruchbedingung der While-Aktivität.

Der Prozessgraph besteht aus zwei Teilgraphen, deren Urbilder bzgl. pg zwei Prozessinstanzmodelle sind, die in Aufrufbeziehung zueinander stehen. Invoke-Aktivität d ruft hierbei den Prozess auf, der nur Invoke-Aktivität e beinhaltet. Die statische Aufrufbeziehung wird im Prozessgraphen durch die invokes-Kante repräsentiert. Der entsprechende Pfeil ⑨ in Abbildung 2.19(b) gehört nicht zur konkreten Syntax der Prozessmodelle und soll hier nur die Aufrufbeziehung verdeutlichen.

Ausführungsinformationen werden im Prozessgraphen in separaten Knoten vorgehalten, im Folgenden "Instanzknoten" genannt in Abgrenzung zu "Definitionsknoten". Eine Aktivitätsinstanz (`ActivityInstance` im SimBPEL-Instance-Metamodel aus Abbildung 2.14) findet sich als Knoten wieder, der entweder gar nicht (entspricht dem Zustand waiting), mit active oder finished beschriftet ist. Über instOf-Kanten verweisen diese Knoten einerseits auf die Definitionsknoten (Aktivitätsknoten), auf die sich die Zustandsinformationen beziehen. So ist der Invoke-Knoten b im Zustand finished Ⓐ. Andererseits verweisen Instanzknoten über belTo-Kanten ("belongs to") auf einen Prozessinstanzknoten (ProcessInstance), der wiederum über eine instOf-Kante einem ProcessDefinition-Knoten zugeordnet ist. Über diese Indirektion können im Prozessgraphen Mehrfachinstanziierungen ausgedrückt werden. Der Übersichtlichkeit halber sind belTo-Kanten in Abbildung 2.19(a) ausgeblendet, sämtliche Instanzknoten verweisen jedoch auf den ProcessInstance-Knoten. Die instOf-Kanten sind ebenfalls nicht sichtbar; Instanzknoten sind in der Unterabbildung direkt adjazent zu ihren jeweiligen Definitionsknoten.

2.5.2 Historiensemantik

Prozessinstanzmodelle beinhalten über `preceded`-Referenzen die Ablaufhistorie des modellierten Prozesses. Eine `preceded`-Referenz besteht zwischen zwei Aktivitätinstanzen i_1 und i_2, wenn i_2 nach i_1 in den Zustand finished gewechselt ist und keine Aktivitätsinstanz j existiert, die nach i_1 und vor i_2 in den Zustand finished übergegangen ist. Die über `preceded`-Referenzen verbundenen `ActivityInstance`-Modellelemente bilden daher eine lineare Liste. Die Zuordnung eines Worts über dem Alphabet aller Aktivitätstypnamen AP' zu einem Prozessinstanzmodell ist daher möglich.

Definition 2.12 (Historienwort) Sei M die Menge aller Prozessinstanzmodelle mit Aktivitäten, deren Aktivitätstypnamen aus der Menge AP' stammen. Sei ferner $r_i(p)$ die Aktivitätsinstanz an i-ter Position in der

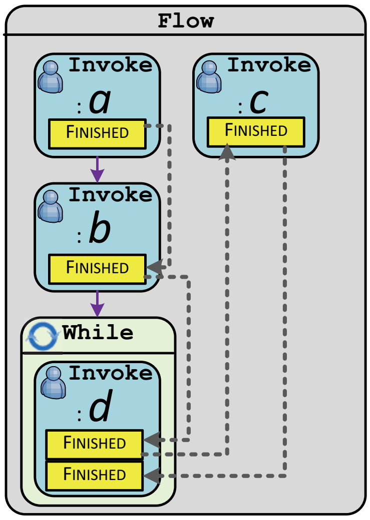


Abbildung 2.20: Prozessinstanzmodell p_{HW} mit Historie

durch `preceded`-Referenzen gebildeten Liste der Länge k im Prozessinstanzmodell $p \in M$ und $t(r_i(p))$ der Typname der zugehörigen Aktivität. Dann ist

$$
\begin{aligned}
h \ &: M \to AP'^*; \\
&p \mapsto t(r_0(p)) \ldots t(r_k(p))
\end{aligned}
$$

die Abbildung, die einem Prozessinstanzmodell ein Wort über AP' zuordnet. Für ein $p \in M$ ist $h(p)$ die *Historien-Semantik* von p.

Beispiel 2.11 (Historienwort) Das Prozessinstanzmodell p_{HW} aus Abbildung 2.20 hat das Historienwort

$$h(p_{HW}) = abdcd.$$

Historienworte lassen sich durch Nachverfolgung von `preceded`-Referenzen (strichlierte Pfeile) in Prozessinstanzmodellen ablesen.

2.5.3 Zukunftssemantik

Graphgrammatiken eignen sich zur Formalisierung der Ausführungssemantik (synonym “Zukunftssemantik”) von Prozessinstanzmodellen. Dies hat folgende

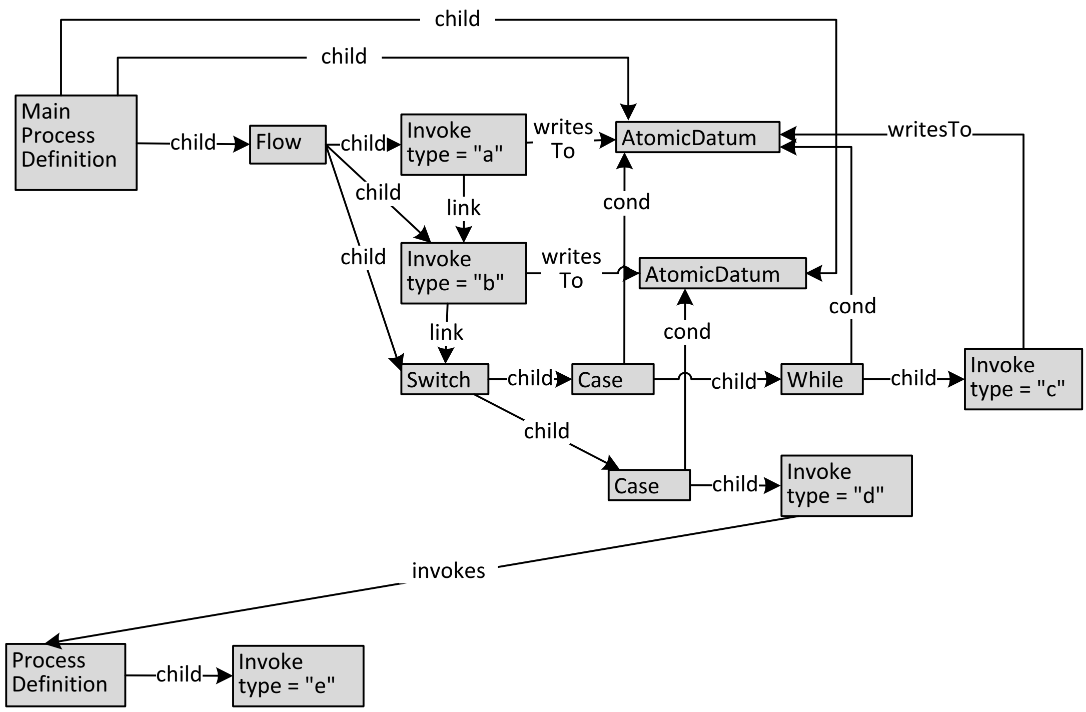


Abbildung 2.21: Startgraph vor Instanziierung (ohne Instanzknoten)

Gründe:

Graphen Ein Prozessinstanzmodell lässt sich in einen Graphen überführen. Ein Prozessgraph $\mathrm{pg}(p)$ eines Prozessinstanzmodells p kann dann als Startgraph einer Graphgrammatik fungieren.

Graphersetzungsregeln Auf Prozessgraphen, die sowohl Ausführungszustände als auch die Ablaufstruktur beinhalten, können Graphersetzungsregeln $\mathcal{R}_{bpel}$ angewendet werden. Diese Regeln $\mathcal{R}_{bpel}$ sind so spezifiziert, dass sie auf Ablaufstruktur und aktuellen Ausführungszustand Rücksicht nehmen. Ein Prozessgraph modelliert also immer ein Ausführungszustand zu einem bestimmten Zeitpunkt; die Graphersetzungsregeln spezifizieren mögliche Zustandsübergänge, die von der im Prozessgraphen festgehaltenen Ablaufstruktur abhängen.

Graphtransitionssystem Ein Prozessgraph $\mathrm{pg}(p)$ bildet zusammen mit der Regelmenge $\mathcal{R}_{bpel}$ eine Graphgrammatik $(\mathcal{R}_{bpel}, \mathrm{pg}(p))$. Das Graphtransitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel}, \mathrm{pg}(p))}$ ist dann die *Zukunftssemantik* des Prozessinstanzmodells p.

Die folgenden Unterunterabschnitte dienen der Erläuterung der Regelmenge $\mathcal{R}_{bpel}$, die die Zukunftssemantik von Prozessinstanzmodellen formal definiert. Beispielhaft wird von dem einfachen Prozessgraphen aus Abbildung 2.21 ausgegangen und die einzelnen Graphersetzungregeln für die

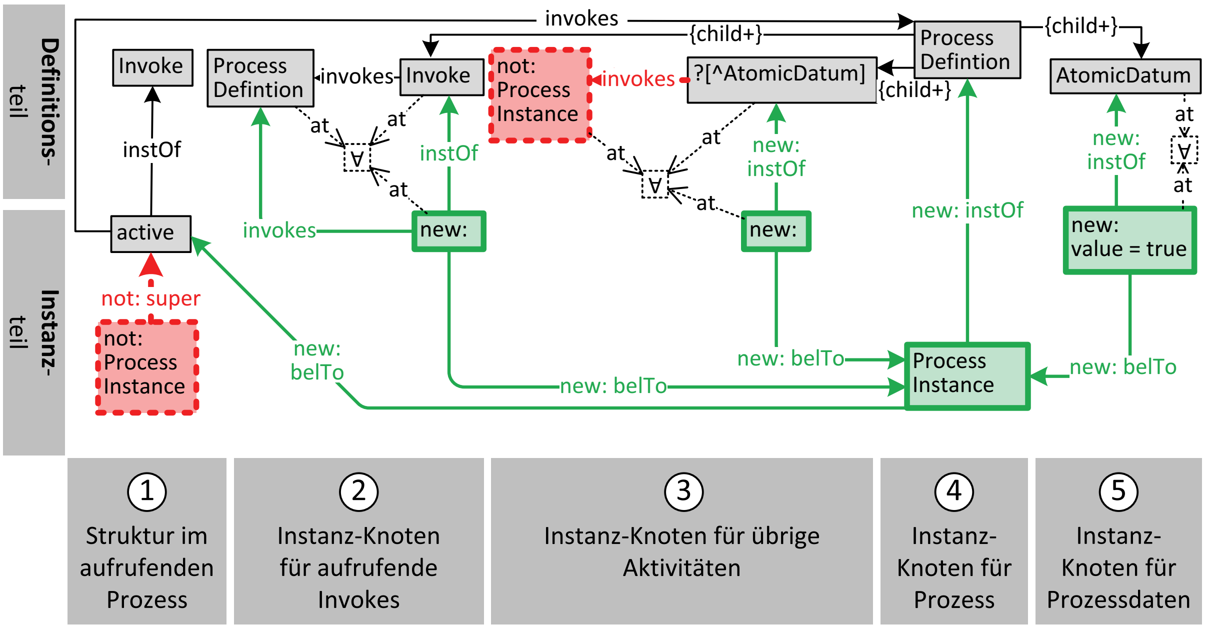


Abbildung 2.22: Graphersetzungsregel instSub für die Instanziierung von Subprozessen

Prozessinstanziierung sowie für Zustandsübergänge von komplexen oder atomaren Aktivitäten beschrieben.

Prozessinstanziierung

Der Prozessgraph in Abbildung 2.21 stellt Prozessdefinitionsmodelle dar, also Prozessinstanzmodelle ohne Ausführungsinformationen. Der Zustand ist in beiden Fällen also der vor der Instanziierung. Diesem Prozessgraphen fehlen daher insbesondere Instanzknoten wie in Abbildung 2.19(a), die Ausführungsinformationen aufnehmen. Daher bezweckt die erste Ersetzungsregel, die auf den Graphen angewendet wird, diese zusätzlichen Knoten zu erzeugen. Die Erzeugung dieser Knoten und zugehöriger Kanten ist der Zweck der instSub benannten Graphersetzungsregel aus Abbildung 2.22.

Die Knoten und Kanten des oberen Regelteils sind der vorhandenen Struktur im Prozessgraphen zuzurechnen, die einem Prozessdefinitionsmodell entspricht. Die Elemente des unteren Teils sind Prozessinstanzen zuzurechnen. Die Regelteile (vertikale Abschnitte der Regel) bewirken Folgendes:

1. In der Anwendungsstelle der Regel muss ein Invoke-Knoten vorhanden sein, der im Zustand active ist und dieser muss mittels invokes-Kante das zu instanziierende Prozessdefinitionsmodell (d.h. dessen Teil im Prozessgraphen) aufrufen. In Regelteil (4) wird ein ProcessInstance-Knoten erzeugt. Der not-Knoten ProcessInstance in Regelteil (1) prüft, ob dieser Knoten bereits existiert und verhindert somit die Mehrfachausführung dieser Regel,

d.h. die unmittelbare Mehrfachinstanziierung.

2. Für alle Invoke-Knoten des instanziierten Prozesses, werden Instanzknoten (unbeschriftet) angelegt. Eventuelle Aufrufbeziehungen (invokes-Kanten) von Unterunterprozessen werden an diese neuen Knoten als Quelle übertragen. So wird zwar die statische Bindung (invokes-Kante zwischen Invoke-Knoten und ProcessDefinition-Knoten in Regelteil (2)) zwischen Prozessen zunächst übernommen, kann aber in jeder Instanz noch dynamisch geändert werden. Wie in Regelteil (1) zu sehen, ist die dynamische Bindung (invokes-Kante zwischen Instanzknoten (active) und ProcessDefinition-Knoten) letztlich ausschlaggebend.

3. Hier werden Instanzknoten für alle übrigen Aktivitätsknoten im instanziierten Prozess angelegt. Das Kriterium, über die diese übrigen Aktivitätsknoten gefunden werden ist, dass sie echte Nachfahren ({child+}) vom ProcessDefinition-Knoten sind und nicht mit AtomicDatum beschriftet, also Aktivitätsknoten, sind und auch keine ausgehende invokes-Kante haben.

4. In diesem Regelteil wird pro Instanziierungsvorgang ein Knoten ProcessInstance angelegt. Dieser verweist per super auf die Aufruferaktivitätsinstanz. Über belTo werden ihm alle Instanzknoten der neuen Instanz zugeordnet.

5. Datenknoten für Prozessdaten (AtomicDatum) werden hier gesondert behandelt. Auch ihnen wird ein Instanzknoten zugeordnet. Diesem wird per Definition bei der Instanziierung der boolesche Wert true zugeordnet.

Die instSub-Regel ist nur für Subprozesse anwendbar, d.h. für Prozesse, die von einem Instanzknoten eines Invoke-Knotens per invokes-Kante aufgerufen werden. Eine hier nicht weiter ausgeführte Variante dieser Regel namens instMain instanziiert den Teil des Prozessgraphen, der den Hauptprozess darstellt. In dieser Regel fehlt im Wesentlichen der Abschnitt (1) aus Abbildung 2.22.

Aktivitätsaktivierung

Instanziierte Prozesse können ausgeführt werden. Im Prozessgraphen heißt das, dass Instanzknoten unter gewissen Voraussetzungen mit active beschriftet werden können. Diese Voraussetzungen sind in der act-Regel aus Abbildung 2.23 formalisiert und lauten informell wie folgt:

1. Alle Vorgänger müssen finished sein, d.h., dass alle Aktivitätsknoten, von denen eine link-Kante zu dem zu aktivierenden Aktivitätsknoten verläuft, müssen in der betreffenden Instanz (ProcessInstance) finished sein.

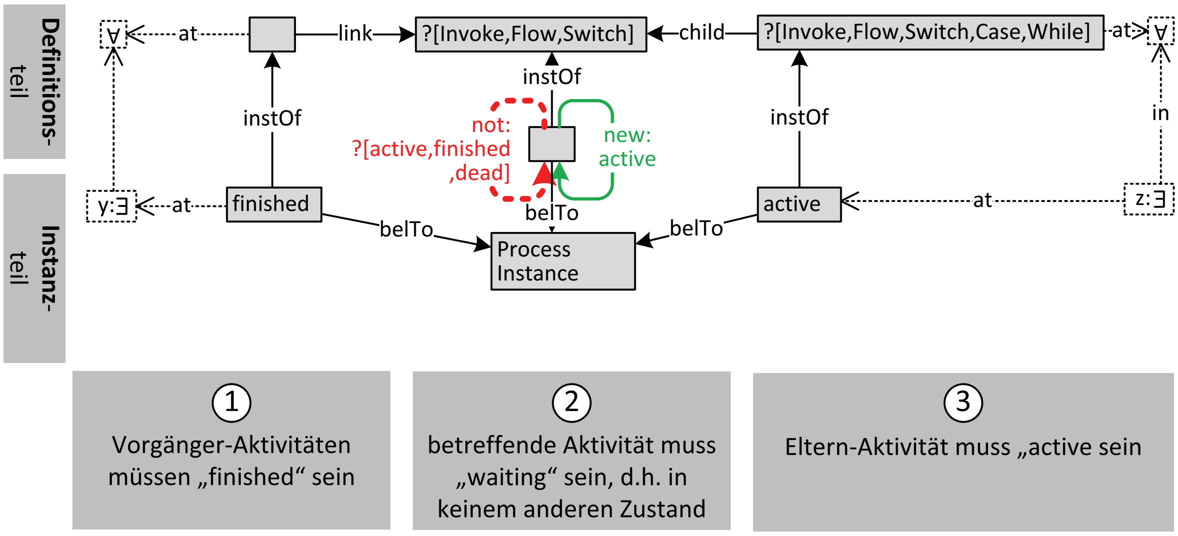


Abbildung 2.23: Graphersetzungsregel act für die Aktivierung von Aktivitätsinstanzen

2. Der Instanzknoten in der betreffenden Instanz darf noch nicht beschriftet sein, was durch die not-Kante ausgedrückt ist. Dies entspricht der Forderung, dass die Aktivitätsinstanz im Zustand waiting ist. Wenn die act-Regel insgesamt anwendbar ist, so wird der Instanzknoten mit active beschriftet.

3. Ist der Aktivitätsknoten Kind (child) eines anderen Aktivitätsknoten, so muss dieser selbst schon active sein.

Die act-Regel "aktiviert" nur Invoke-, Flow-, und Switch-Aktivitätsknoten. Für die Aktivierung von Case- und While-Aktivitätsknoten sind separate Regeln vorhanden. Die Regel actCase ist nur anwendbar, wenn der Instanzknoten des AtomicDatum-Knotens den Wert true hat, der per cond-Kante mit dem betreffenden Case-Knoten verbundenen ist, die dargestellte Zweigbedingung also erfüllt ist.

Die Regel actWhile gleicht act bis auf die zusätzliche Bedingung, dass die per cond verbundene Schleifenbedingung wahr sein muss und noch kein Schleifendurchlauf stattgefunden hat bzw. der vorherige Schleifendurchlauf abgeschlossen ist.

Aktivitätsinvalidierung

Zu den Regeln actCase und actWhile gibt es komplementäre Regeln killCase und killWhile. Diese sind in Situationen anwendbar, wenn actCase bzw. actWhile nur deshalb nicht anwendbar sind, weil die mit cond referenzierte Zweig- bzw. Schleifenbedingung false ist. In diesem Fall wird der betreffende Case- bzw. While-Knoten nicht mit active sondern mit dead beschriftet. Eine

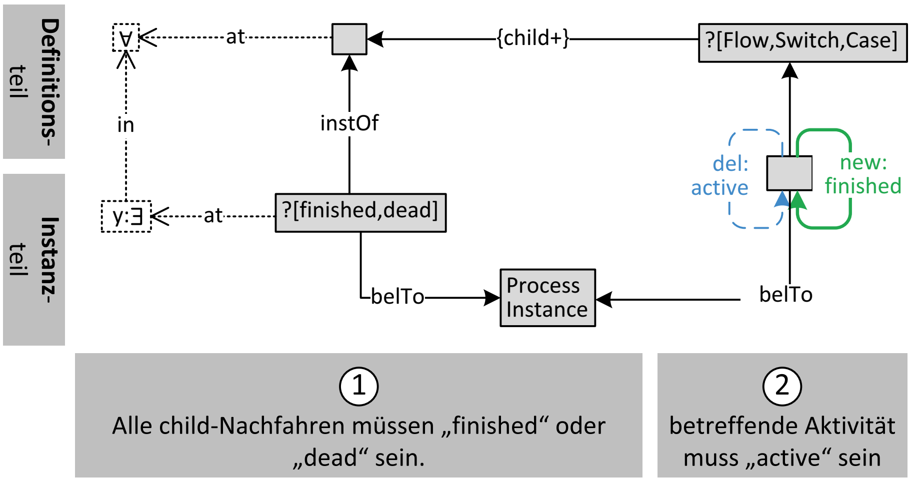


Abbildung 2.24: Graphersetzungsregel fin für Finalisierung aktiver Aktivitätsknoten

zusätzliche Regel propDead sorgt für die Propagation des dead-Zustandes auf die child-Nachkommen toter Aktivitätsknoten.

Aktivitätsfinalisierung

Die Graphersetzungsregel fin aus Abbildung 2.24 sorgt für die Finalisierung aktiver Flow-, Switch- und Case-Aktivitätsknoten. Die Kriterien hier sind:

1. Instanzknoten von child-Nachkommen des betreffenden Aktivitätsknoten müssen in einem finalen Zustand sein, d.h. entweder finished oder dead.

2. Der Instanzknoten des betreffenden Aktivitätsknotens muss mit active beschriftet sein. Dann wird diese Beschriftung durch finished ersetzt.

Invoke-Aktivitätsknoten werden durch die finInvoke-Graphersetzungsregel finalisiert. In dieser wird zusätzlich zu fin gefordert, dass ein evtl. aufgerufener Prozess bereits beendet ist, d.h. dass dessen Aktivitäten alle in einem finalen Zustand sind.

In der Regel finWhile wird gefordert, dass alle child-Nachfahren in einem finished-Zustand sind, die Schleife also mindestens einmal durchlaufen wurde und die Schleifenbedingung jetzt false ist.

Schleifenreiteration

Schleifen können mehrfach durchlaufen werden. Eine Regel resetWhile manipuliert im Prozessgraphen Instanzknoten von child-Nachfahren eines While-Knotens so, dass sie wieder wie vor der ersten Iteration alle ohne Beschriftung, d.h. im Zustand waiting sind. Die Anwendbarkeit von resetWhile hängt von folgenden drei Bedingungen ab:

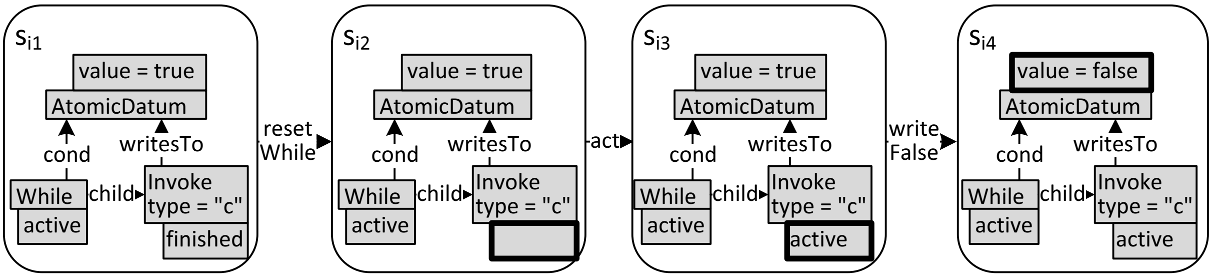


Abbildung 2.25: Anwendungen von resetWhile und writeFalse

1. Der Instanzknoten eines While-Knotens w ist im Zustand active
2. Alle Instanzknoten der child-Nachfahren von w sind im Zustand finished oder dead. Wenn das der Fall ist, sind also alle Aktivitäten im Schleifenrumpf abgeschlossen.
3. Der While-Knoten w verweist per cond-Kante (immer noch) auf einen Datenknoten mit Beschriftung true. Die Schleifenbedingung ist also zu diesem Zeitpunkt erfüllt.

Beispiel 2.12 (Anwendung von resetWhile) Abbildung 2.25 zeigt Prozessgraphen, die Fragmente des Prozessgraphen aus Abbildung 2.19(a) sind. Beim Übergang von s_{i1} zu s_{i2} durch Anwendung von resetWhile wird die Beschriftung des Instanzknotens des einzigen child-Nachfahrens des While-Knotens gelöscht.

An dieser Stelle zeigt sich eine bewusst gewählte Abweichung zwischen einem Prozessinstanzmodell p und dem zugehörigen Prozessgraphen $\mathrm{pg}(p)$: Bei Mehrfachdurchläufen von Schleifen innerhalb eines Prozessinstanzmodells p werden ggf. mehrere Aktivitätsinstanzen einer Aktivität im Schleifenrumpf zugeordnet. Es wäre daher naheliegend, pro Schleifendurchlauf im Prozessgraphen $\mathrm{pg}(p)$ neue, unbeschriftete Instanzknoten anzulegen, statt die Beschriftungen finished oder dead der alten zu löschen. Dies führt jedoch dazu, dass sich der Prozessgraph nach jeder Iteration in jedem Fall von allen anderen bisherigen Prozessgraphen unterscheidet. Da eine Schleife prinzipiell beliebig häufig durchlaufen werden kann, ergeben sich daher allein durch das Vorhandensein eines While-Knotens auch beliebig viele Prozessgraphen. Daher würde ein erzeugtes Graphtransitionssystem beliebig viele Zustände besitzen, d.h. die Zustandsmenge unbeschränkt sein. Die spätere Verwendung von Graphtransitionssystemen, die mittels der Regelmenge $\mathcal{R}_{bpel}$ erzeugt werden, erfordert jedoch eine endliche Transitionssystemgröße.

Zugriffe auf Prozessdaten

Werte von AtomicDatum-Knoten können über die Regeln writeTrue und writeFalse geändert werden. Diese Regeln sind anwendbar, wenn einer der Instanzknoten eines Aktivitätsknotens a mit active beschriftet ist und a über eine writesTo-Kante auf einen AtomicDatum-Knoten d verweist. Dann wird die Beschriftung des Instanzknoten von d auf true bzw. false geändert.

Beispiel 2.13 (Anwendung von writeFalse) In Abbildung 2.25 wird beim Übergang von s_{i3} zu s_{i4} die Regel writeFalse angewendet. Hierdurch wird insbesondere die Schleifenbedingung unwahr. Somit kann nach s_{i4} die Schleife nicht nochmals durchlaufen werden.

2.6 Semantikdefinition für Prozesswissensmodelle

Im vorherigen Abschnitt wurden die Semantiken (Zustand, Historie und Zukunft) von SimBPEL-Instance-Prozessinstanzmodellen dargelegt. Darauf aufbauend wird nun die Semantik von BPCL-Prozesswissensmodellen formal definiert. Dies geschieht indirekt über die schrittweise Formalisierung der Komplianzbeziehung

$$p \models \mathcal{B}$$

zwischen einem Prozessinstanzmodell p und einem Prozesswissensmodell $\mathcal{B}$ (① in Abbildung 2.26).

Hierbei ist zunächst ein BPCL-Prozesswissensmodell $\mathcal{B}$ als syntaktisch beliebig komplexes Objekt mit explizit und durch Spezialisierungen implizit modellierten Bedingungsbeziehungen in eine boolesche Konjunktion zu überführen (② in Abbildung 2.26), deren atomare Prädikate wiederum Komplianzbeziehungen sind, jedoch zwischen dem Prozessinstanzmodell p und jeweils einer Bedingungsbeziehung, also von der Form

$$p \models \left(b \xrightarrow[n..m]{Art} a \right). \quad ③$$

In einem weiteren Schritt ④ wird jeder Bedingungsbeziehung $b \xrightarrow[n..m]{Art} a$ ein komplexer formaler Ausdruck zugeordnet, der Aussagen über die Historien- und Zukunftssemantik des Prozessinstanzmodells p miteinander verknüpft.

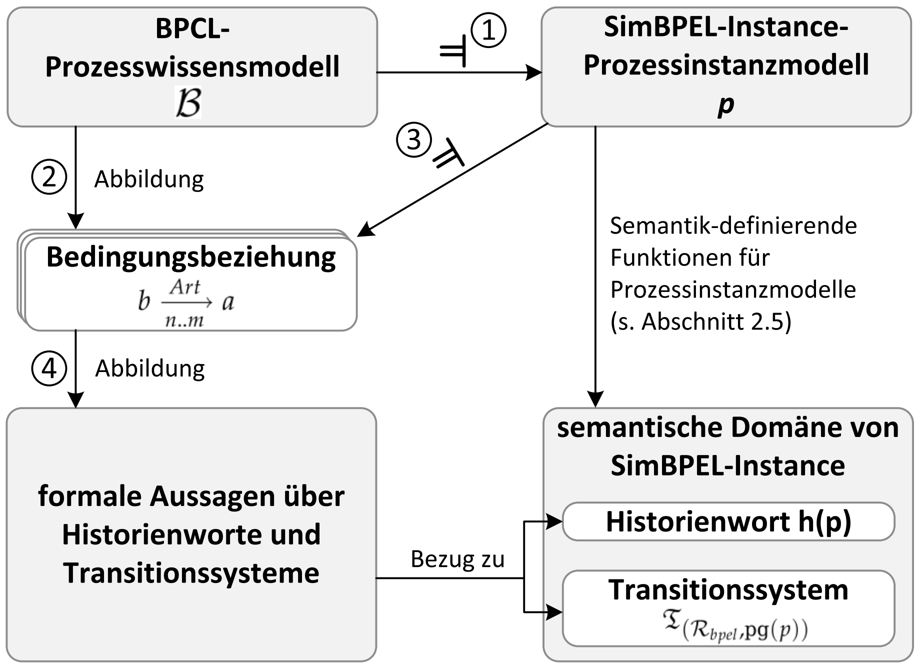


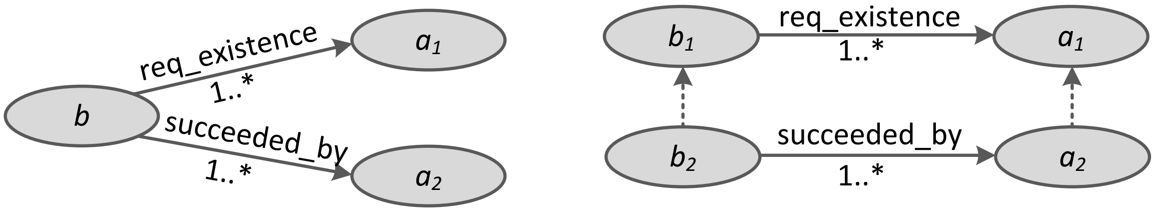
Abbildung 2.26: Zusammenhänge bei der Semantikdefinition von Prozesswissensmodellen

2.6.1 Bedingungsbeziehungen eines Prozesswissensmodells

Bedingungsbeziehungen werden in BPCL-Prozesswissensmodellen explizit oder implizit modelliert. Die Frage, ob ein Prozessinstanzmodell p kompliant zu einem Prozesswissensmodell $\mathcal{B}$ ist, lässt sich nur darüber klären, ob p bestimmte Bedingungsbeziehungen erfüllt. Hierbei spielen insbesondere Spezialisierungsbeziehungen in $\mathcal{B}$ eine Rolle, die zu impliziten Bedingungsbeziehungen führen oder die Menge aller Aktivitätstypen erweitern, die zur Erfüllung einer Bedingungsbeziehung beitragen können.

Die folgenden Erläuterungen haben zum Ziel, die Semantik von BPCL-Prozesswissensmodellen zu klären. Dabei wird ein Prozesswissensmodell zunächst auf seine in ihm vorhandenen Bedingungsbeziehungen reduziert. Die Semantik einer einzelnen Bedingungsbeziehung wird dann in Unterabschnitt 2.6.2 formal definiert.

Das folgende Beispiel zeigt den Zusammenhang zwischen explizit in einem graphischen BPCL-Prozesswissensmodell modellierten Bedingungsbeziehungen und einer mathematischen Beschreibung $\mathrm{expBB}(\mathcal{B})$ für diese Menge von Bedingungsbeziehungen. Auf eine weitere Formalisierung von $\mathrm{expBB}(\mathcal{B})$ wird verzichtet, da diese nur zwischen den unterschiedlichen Darstellungsformen von explizit modellierten Bedingungsbeziehungen übersetzt, nämlich zwischen solchen in graphischen BPCL-Prozesswissensmodellen und zwischen



(a) Prozesswissensmodell $\mathcal{B}_1$ ohne Spezialisierungen

(b) Prozesswissensmodell $\mathcal{B}_2$ mit Spezialisierungen

Abbildung 2.27: Beispielhafte Prozesswissensmodelle

Mengen von Bedingungsbeziehungen in textueller Notation.

Bedingungsbeziehungen können innerhalb von Prozesswissensmodellen graphisch und explizit notiert werden. Die folgende Definition vermittelt semiformal zwischen den expliziten, graphisch notierten Bedingungsbeziehungen in einem Prozesswissensmodell $\mathcal{B}$ und der bereits zuvor gebrauchten mathematischen und textuellen Notation $b \xrightarrow[n..m]{Art} a$.

Beispiel 2.14 (Explizite Bedingungsbeziehung) Für die beispielhaften Prozesswissensmodelle $\mathcal{B}_1$ und $\mathcal{B}_2$ aus Abbildung 2.27(a) bzw. Abbildung 2.27(b) gilt

➢ $\text{expBB}(\mathcal{B}_1) = \{b \xrightarrow[1..*]{req\_existence} a_1, b \xrightarrow[1..*]{succeeded\_by} a_2\}$ und

➢ $\text{expBB}(\mathcal{B}_2) = \{b_1 \xrightarrow[1..*]{req\_existence} a_1, b_2 \xrightarrow[1..*]{succeeded\_by} a_2\}$.

Besitzt ein Prozesswissensmodell $\mathcal{B}$ keine Spezialisierungsbeziehungen, ist ein Prozessinstanzmodell p zu $\mathcal{B}$ kompliant gdw. p zu allen explizit modellierten Bedingungsbeziehungen von $\mathcal{B}$ kompliant ist. (Implizit modellierte Bedingungsbeziehungen existieren in $\mathcal{B}$ nicht.) Formal bedeutet dies: Es gilt $p \models \mathcal{B}$ genau dann, wenn

$$\bigwedge_{\left(b \xrightarrow[n..m]{Art} a\right) \in \text{expBB}(\mathcal{B})} p \models b \xrightarrow[n..m]{Art} a$$

gilt.

Das BPCL-Prozesswissensmodell $\mathcal{B}_1$ aus Abbildung 2.27(a) modelliert zwei

Bedingungsbeziehungen. Es gilt $p \models \mathcal{B}_1$ genau dann, wenn

$$p \models b \xrightarrow[1..*]{req\_existence} a_1 \quad \wedge \quad p \models b \xrightarrow[1..*]{succeeded\_by} a_2$$

gilt.

Es müssen also beide ($\wedge$) Bedingungsbeziehungen erfüllt sein, damit p kompliant zu $\mathcal{B}_1$ ist ($p \models \mathcal{B}_1$).

Die Überführung eines Prozesswissensmodells $\mathcal{B}$ mit Spezialisierungen in boolesch verknüpfte Bedingungsbeziehungen ist komplizierter, da Spezialisierungen sowohl Quell- als auch Zielaktivitäten betreffen können und Mengen von Quell- und Zielaktivitätstypen einführen, die zunächst erst wieder auf Bedingungsbeziehungen mit einelementiger Quell- und Zielseite reduziert werden müssen.

Definition 2.13 (Subaktivitätstypen eines Typs) Für ein Prozesswissensmodell $\mathcal{B}$ und einen darin enthaltenen Aktivitätstypen $a \in AT$ sei

$$\text{subAT}(a) \subseteq AT$$

die Menge aller Aktivitätstypen, die a direkt oder indirekt spezialisieren. Insbesondere gelte immer $a \in \text{subAT}(a)$.

Beispielsweise gilt für das Prozesswissensmodell $\mathcal{B}_2$ aus Abbildung 2.27(b):

$$\text{subAT}(b_1) = \{b_1, b_2\}.$$

Ein Aktivitätstyp b', der einen b spezialisiert, erbt alle Bedingungsbeziehungen, für die b der Quelltyp ist. Formal ausgedrückt bedeutet das, dass

$$p \models \text{subAT}(b) \xrightarrow[n..m]{Art} \text{subAT}(a)$$

per Definition genau dann gilt, wenn

$$\bigwedge_{b' \in \text{subAT}(b)} p \models b' \xrightarrow[n..m]{Art} \text{subAT}(a)$$

gilt.

Die Menge auf Zielseite kann nicht so trivial aufgelöst, d.h. auf Bedingungsbeziehungen mit einelementiger Zielseite, reduziert werden, da zur Erfüllung einer Bedingungsbeziehung $b \xrightarrow[n..m]{Art} a$ unterschiedliche Subtypen von a gemeinsam beitragen können. Zunächst ist es praktisch, eine Bedingungsbeziehung nach Unter- und Obergrenze aufzutrennen. Es gilt

$$p \models b \xrightarrow[n..m]{Art} \text{subAT}(a)$$

genau dann, wenn

$$p \models b \xrightarrow[n..*]{Art} \text{subAT}(a) \land p \models b \xrightarrow[0..m]{Art} \text{subAT}(a)$$

gilt.

Sei ferner $\text{subAT}(a) = \{a_1, \ldots, a_k\}$. Dann gilt

$$p \models b \xrightarrow[n..*]{Art} \text{subAT}(a)$$

per Definition genau dann, wenn

$$\exists n_1 \ldots \exists n_k : \left(\sum_{1 \leq i \leq k} n_i = n \right) \land \; p \models b \xrightarrow[n_1..*]{Art} a_1 \;\land\; \ldots \;\land\; p \models b \xrightarrow[n_k..*]{Art} a_k$$

gilt. Des Weiteren gilt

$$p \models b \xrightarrow[0..m]{Art} \text{subAT}(a)$$

per Definition genau dann, wenn

$$\exists m_1 \ldots \exists m_k : \left(\sum_{1 \leq i \leq k} m_i = m \right) \land \; p \models b \xrightarrow[0..m_1]{Art} a_1 \;\land\; \ldots \;\land\; p \models b \xrightarrow[0..m_k]{Art} a_k$$

gilt. Zusammenfassend kann dann nun Folgendes definiert werden:

Definition 2.14 (Bedingungsbeziehungen in Prozesswissensmod.) Für ein Prozessinstanzmodell p und ein Prozesswissensmodell $\mathcal{B}$ sowie $\text{expBB}(\mathcal{B})$ und $\text{subAT}(x)$ wie zuvor definiert gilt $p \models \mathcal{B}$ per Definition genau dann, wenn

$$\bigwedge_{b \xrightarrow[n..m]{Art} a \in \text{expBB}(\mathcal{B})} p \models \left(\text{subAT}(b) \xrightarrow[n..m]{Art} \text{subAT}(a) \right)$$

gilt.

2.6.2 Semantikdefinition für Bedingungsbeziehungen

Die Semantik von Komplianzbeziehungen $p \models b \xrightarrow[n..m]{Art} a$ zwischen einem Prozessinstanzmodell und einer Bedingungsbeziehung wird mit Bezug auf die Historiensemantik $h(p)$ und Zukunftssemantik $\mathfrak{T}_{\mathcal{R}_{bpel},\mathrm{pg}(p)}$ von p definiert. Für Letzteres wird die erweiterte Computational Tree Logic verwendet und daher im Folgenden zunächst syntaktisch und semantisch erläutert bevor die Semantik von Bedingungsbeziehungen formalisiert wird.

Extended Computational Tree Logic (CTL)*

Die *erweiterte Computational Tree Logic (CTL*)* gehört zu den Temporalen Logiken, deren Ausdrücke auf Transitionssystemen ausgewertet werden. Die folgenden Definitionen der Syntax und Semantik von CTL* sind an die in [CGP99, Seite 28f] angelehnt.

Definition 2.15 (Syntax von CTL*) In CTL* wird zwischen Zustandsformeln f und Pfadformeln g unterschieden. Die Sprache von CTL* ist die Menge aller *Zustandsformeln*. Die Menge aller Zustandsformeln ist induktiv mittels einer Grammatik in vereinfachter EBNF wie folgt definiert:

$$\begin{aligned} f := {} & p \in AP \mid \top \mid \bot && \text{(atomare Aussagen, true und false)} \\ & \mid \neg f \mid f \vee f \mid f \wedge f && \text{(boolesche Operatoren)} \\ & \mid \mathbf{E}g \mid \mathbf{A}g && \text{(Pfadquantoren)} \end{aligned}$$

Die Menge aller *Pfadformeln* ist definiert als

$$\begin{aligned} g := {} & f && \text{(Zustandsformeln)} \\ & \mid \neg g \mid g \vee g \mid g \wedge g && \text{(boolesche Operatoren)} \\ & \mid \mathbf{X}g \mid \mathbf{F}g \mid \mathbf{G}g \mid g\mathbf{U}g && \text{(temporale Operatoren)} \end{aligned}$$

Vermittels der Ableitung $g := f$ sind Zustandsformeln also gleichzeitig auch Pfadformeln (aber nicht umgekehrt).

Beispiel 2.15 (CTL*-Formeln) Seien a und b atomare Aussagen. Dann sind gemäß den Ableitungsregeln die Wörter $\mathbf{AGAF}a$ und $\mathbf{EXXX}a \vee \mathbf{AF}b$ CTL*-Formeln. $\mathbf{XA}a$ hingegen ist keine CTL*-Formel, da aus den Ableitungsregeln folgt, dass ein temporaler Operator nicht das erste Zeichen sein darf.

Im Folgenden werden immer Formeln aus Teilsprachen von CTL* verwendet. Diese Teilsprachen sind LTL (Linear Temporal Logic) und CTL (Computational Tree Logic) [Bra85]. Die Syntax ist gegenüber CTL* eingeschränkt.

Definition 2.16 (Syntax von CTL) Die Regel

$$g := f$$

aus der Grammatik für CTL*-Formeln ist zu ersetzen durch

$$g := \ \mathbf{X}f \mid \mathbf{F}f \mid \mathbf{G}f \mid f\mathbf{U}f.$$

CTL-Formeln sind also dadurch charakterisiert, dass einem temporalen Operator immer unmittelbar ein Pfadquantor vorausgehen muss.

Definition 2.17 (Syntax von LTL) Die Menge aller Pfadformeln f ist

$$f := a \in AP \mid \neg f \mid f \vee f \mid f \wedge f \mid \mathbf{X}f \mid \mathbf{F}f \mid \mathbf{G}f \mid f\mathbf{U}f.$$

Die Menge aller Zustandsformeln g ist

$$g := \mathbf{A}(f).$$

LTL-Formeln beginnen also immer mit einem Pfadquantor **A**; daraufhin folgt eine Pfadformel bestehend nur aus temporalen Operatoren, booleschen Operatoren und atomaren Aussagen. Die Sprachen LTL und CTL sind überlappend, enthalten einander nicht, sind aber beide echt in CTL* enthalten.

Semantik von CTL* Die Semantik von CTL* kann im Gegensatz zur Syntax nicht isoliert erklärt werden, sondern nimmt Bezug auf Pfade und Zustände in (Graph-)Transitionssystemen (vgl. Definition 2.5).

Atomare Aussagen in Graphtransitionssystemen sind Graphersetzungsregeln (vgl. Unterabschnitt 2.4.2). Das heißt, dass innerhalb eines Graphtransitionssystems in einem Zustand s die atomare Aussage a gilt, wenn a (als

Graphersetzungsregel) auf den Graphen s passt. Innerhalb von CTL*-Formeln steht also der Name der jeweiligen Graphersetzungsregel für eine atomare Aussage. Neben den Regelnamen können atomare Aussagen Parameter besitzen, die auf Beschriftungen von am Matching beteiligter Knoten hinweisen.

Definition 2.18 (Semantik von CTL*)
Die Semantik der Sprachkonstrukte von CTL* ist induktiv wie folgt definiert, wobei a wie zuvor für atomare Aussagen, $f_{(i)}$ für Zustandsformeln und $g_{(i)}$ für Pfadformeln gemäß Definition 2.16, s für Zustandsgraphen und π für Sequenzen benachbarter Zustände (Pfade) stehen (vgl. Definition 2.10):

$$M, s \models a \iff a \in L(s) \tag{2.1}$$
$$M, s \models \neg f \iff M, s \not\models f \tag{2.2}$$
$$M, s \models f_1 \vee f_2 \iff M, s \models f_1 \text{ oder } M, s \models f_2 \tag{2.3}$$
$$M, s \models f_1 \wedge f_2 \iff M, s \models f_1 \text{ und } M, s \models f_2 \tag{2.4}$$
$$M, s \models \mathbf{E}g \iff \exists \pi : \pi_0 = s \wedge M, \pi_1 \models g \tag{2.5}$$
$$M, s \models \mathbf{A}g \iff \forall \pi : \pi_0 = s \Rightarrow M, \pi_1 \models g \tag{2.6}$$
$$M, \pi \models f \iff \pi_0 = s \Rightarrow M, s \models f \tag{2.7}$$
$$M, \pi \models \neg g \iff M, \pi \not\models g \tag{2.8}$$
$$M, \pi \models g_1 \vee g_2 \iff M, \pi \models g_1 \text{ oder } M, \pi \models g_2 \tag{2.9}$$
$$M, \pi \models g_1 \wedge g_2 \iff M, \pi \models g_1 \text{ und } M, \pi \models g_2 \tag{2.10}$$
$$M, \pi \models \mathbf{X}g \iff M, \pi_1 ... \models g \tag{2.11}$$
$$M, \pi \models \mathbf{F}g \iff \exists k \in \mathbb{N}_0 : M, \pi_k ... \models g \tag{2.12}$$
$$M, \pi \models \mathbf{G}g \iff \forall i \in \mathbb{N}_0 : M, \pi_i ... \models g \tag{2.13}$$
$$M, \pi \models g_1 \mathbf{U} g_2 \iff \exists k \in \mathbb{N}_0 : (M, \pi_k ... \models g_2 \tag{2.14}$$
$$\wedge \forall j \in \mathbb{N}_0 : j < k \Rightarrow M, \pi_j ... \models g_1) \tag{2.15}$$

Der Zusammenhang zwischen Pfad- und Zustandsformeln wird dabei in Zeile 2.7 hergestellt.

Semantik für Bedingungsbeziehungen

Die Semantik von Komplianzbeziehungen $p \models b \xrightarrow[n..m]{Art} a$ wird durch Historienworte und CTL*-Ausdrücke definiert, die sich auf Graphtransitionssysteme beziehen, die durch die Graphgrammatik aus Unterabschnitt 2.5.3 erzeugt werden.

Die atomaren Aussagen in den CTL*-Formeln beziehen sich auf die Finalisierung von Invoke-Aktivitäten, d.h. eine atomare Aussage a soll in einem Zustand s' gelten, wenn dieser Zustand durch einen Übergang finInvoke(a) erreicht wurde, d.h. im Transitionssystem ein Übergang $s \xrightarrow{\text{finInvoke(a)}} s'$ für einen Zustand s existiert.

Nun sind in Graphtransitionssystemen atomare Aussagen in einem Zustand immer in diesem Zustand passende Regelanwendungen, aber nicht Regelanwendungen, über die dieser Zustand erreicht wurde. Für s' kann finInvoke(a) also keine atomare Aussage sein. Daher bedarf es einer Erweiterung der in Abschnitt 2.5 erläuterten Graphgrammatik. Diese Erweiterung ist eine Graphersetzungsregel lfa, die immer auf Zustände angewendet wird, in die mittels finInvoke gewechselt wurde. In diesen Zuständen ist dann allein lfa(a) anwendbar und zwar genau einmal. Insgesamt wird durch die Erweiterung im erzeugten Graphtransitionssystem jeder Übergang $s \xrightarrow{\text{finInvoke(a)}} s'$ erweitert zu $s \xrightarrow{\text{finInvoke(a)}} s'' \xrightarrow{\text{lfa(a)}} s'$, wobei lfa(a) der einzige Übergang ist, der s'' als Quelle hat.

Der Parameter a gibt dabei den Typ (type-Datenknoten) der zuvor finalisierten Invoke-Aktivität an. Die Anzahl der Zustände auf einem Pfad im Graphtransitionssystem, in denen lfa(a) gilt, ist also gleich der Anzahl der Finalisierungen von Invoke-Aktivitäten vom Typ x. Der Übersichtlichkeit halber werden in den nachfolgenden CTL*-Formeln atomare Aussagen nur mit dem jeweiligen Parameter bezeichnet, d.h. ein a steht für die atomare Aussage lfa(a). Im Weiteren sei die Menge der relevanten atomaren Aussagen AP' gerade so gewählt, dass sie nur lfa-Graphersetzungsregeln beinhaltet. Unter Berücksichtigung der Abkürzung ist für das Beispiel aus Abbildung 2.19(a) also $AP' = \{a, b, c, d, e\}$.

Die Semantikdefinition geschieht vorzugsweise über CTL und auch dort, wo auch eine Definition in LTL möglich wäre. Der Grund hierfür ist, dass GROOVE in der Lage ist, Formeln in CTL auf erzeugten Graphtransitionssystemen auszuwerten. Die folgenden, komplizierten Semantikdefinitionen in CTL konnten so mit GROOVE überprüft werden.

Die Semantikdefinitionen sind aufeinander aufgebaut. Jede Bedingungsbeziehungsart wird zunächst für eine variable Obergrenze m (und feste, minimale Untergrenze 0) danach für eine variable Untergrenze n (und feste, maximale Obergrenze $*$) definiert. Die Semantikdefinition für den allgemeinen Fall $n..m$ wird dann durch eine boolesche Konjunktion hergestellt.

Die Semantikdefinitionen sind Äquivalenzen; ihre Struktur ist allgemein

$$p \models b \xrightarrow[n..m]{Art} a \;:\Longleftrightarrow\; S(Art, n, m),$$

wobei $S(Art, n, m)$ ein mathematisch-logischer Ausdruck ist, der von der Bedingungsbeziehungsart (Art) und den Multiplizitätsgrenzen n und m abhängt.

Das Prozessinstanzmodell p ist also kompliant zu $b \xrightarrow[n..m]{Art} a$ genau dann, wenn $S(Art, n, m)$ für p gilt. In $S(Art, n, m)$ gelte der Lesbarkeit halber die Abkürzung

$$\mathfrak{T}_p := \mathfrak{T}_{(\mathcal{R}_{bpel}, \mathrm{pg}(p))}.$$

$S(Art, n, m)$ stützt sich außerdem auf die Historiensemantik $h(p)$ ab, die bereits in Unterabschnitt 2.5.2 geklärt wurde. Des Weiteren ist $S(Art, n, m)$ in Klauseln gegliedert. Die Reihenfolge der Klauseln ist typischerweise derart, dass die erste Klausel Bedingungen an die Prozesszukunft formuliert, die zweite an die Prozesshistorie und eine ggf. vorhandene dritte Implikationen zwischen Historie und Zukunft. Die Klauseln sind logisch konjugiert, wobei der besseren Lesbarkeit halber “und” bzw. “oder” ausgeschrieben wurde.

includes-Semantik

Im Folgenden sind Semantikdefinitionen für includes-Ausdrücke angegeben. Es wird für diese Bedingungsbeziehung vorausgesetzt, dass der Quellaktivitätstyp b immer nur für oberste Aktivitäten in Prozessinstanzmodellen bzgl. Komposition angewendet wird.

Definition 2.19 (includes-Semantik für 0..m) Es gelte $p \models b \xrightarrow[0..m]{includes} a$ genau dann, wenn

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \neg\, \underbrace{(\mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \ldots)}_{m-|h(p)|_a+1 \text{ mal}} \ldots)$$

$$\text{und}$$

$$|h(p)|_a \leq m.$$

gilt. Die Äquivalenz besagt, dass bei einer Obergrenze m zum einen in der Prozesshistorie $h(p)$ nicht mehr als m-mal a vorkommen dürfen. Des Weiteren dürfen in der Prozesszukunft $\mathfrak{T}_p$ nicht $m - |h(p)|_a + 1$ mal a vorkommen.

Angenommen, $m - |h(p)|_a + 1 = 2$. Dann nimmt die CTL-Formel folgende Form an:

$$\neg\mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \mathbf{EXEF}a))$$

Definition 2.20 (includes-Semantik für $n..*$) Es gelte $p \models b \xrightarrow[n..*]{includes} a$ genau dann, wenn

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \underbrace{\mathbf{AXAF}(a \wedge \mathbf{AXAF}(a \wedge \ldots)}_{n-|h(p)|_a \text{ mal}} \ldots)$$

oder

$$|h(p)|_a \geq n$$

gilt. Hier ist der bzgl. der Äquivalenz rechtsseitige Ausdruck so zu verstehen, dass für den Fall $|h(p)|_a < n$ die verbleibenden a in der Prozesszukunft vorkommen oder bereits mindestens n-mal a in der Prozesshistorie vorkommt ($|h(p)|_a \geq n$).

Beispielsweise für den Spezialfall $n - |h(p)|_a = 2$ lautet die CTL-Formel wie folgt:

$$\mathbf{AXAF}(a \wedge \mathbf{AXAF}a)$$

Definition 2.21 (includes-Semantik für $n..m$) Es gelte $p \models b \xrightarrow[n..m]{includes} a$ genau dann, wenn

$$p \models b \xrightarrow[0..m]{includes} a \wedge \; p \models b \xrightarrow[n..*]{includes} a$$

gilt.

preceded_by-Semantik

Die folgenden Definitionen beziehen sich auf Aktivitäten unterschiedlichen Typs a und b. Gemein ist den definierenden Ausdrücken, dass sie insbesondere dann wahr werden, wenn b nicht ausgeführt wird bzw. ausgeführt worden ist. Das ist insbesondere dann der Fall, wenn im Prozess keine b-Aktivität existiert.

Definition 2.22 (preceded_by-Semantik für 0..m) Es gelte $p \models b \xrightarrow[0..m]{preceded\_by} a$ genau dann, wenn

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \neg \underbrace{\mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \ldots \mathbf{EXEF}(a}_{m-|h(p)|_a+1 \text{ mal}} \wedge \mathbf{EXEF} b) \ldots)$$

und

$$\forall k:\ h(p)_k = b \ \Rightarrow\ |h(p)_0 \ldots h(p)_k|_a \leq m$$

gilt. Wenn also in der Prozesshistorie an Position k ein b durchgeführt wird, so darf vorher maximal m-mal a ausgeführt werden. Die CTL-Formel besagt, dass die $m - |h(p)|_a + 1$-fache Ausführung von a vor einem b in der Prozesszukunft die Bedingung verletzt.

Für den Spezialfall $m - |h(p)|_a + 1 = 2$ lautet die CTL-Formel

$$\neg \mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \mathbf{EXEF} b))).$$

Definition 2.23 (preceded_by-Semantik für $n..*$) Es gelte $p \models b \xrightarrow[n..*]{preceded\_by} a$ genau dann, wenn

$$\begin{gathered} \mathfrak{T}_p, \mathrm{pg}(p) \models \neg \mathbf{E}(\neg a \mathbf{U} b) \\ \wedge \neg \mathbf{E}(\neg a \mathbf{U}(a \wedge \mathbf{EXE}(\neg a \mathbf{U} b))) \\ \ldots \\ \wedge \underbrace{\neg \mathbf{E}(\neg a \mathbf{U}(a \wedge \mathbf{EXE}(\neg a \mathbf{U}(a \wedge \ldots \mathbf{EXE}(a \mathbf{U} b)}_{n-|h(p)|_a \text{ mal}} \ldots) \end{gathered}$$

und

$$\forall k:\ h(p)_k = b \ \Rightarrow\ |h(p)_0 \ldots h(p)_k|_a \geq n$$

gilt.

Für den Spezialfall $n - |h(p)|_a = 2$ ist die CTL-Formel

$$\neg \mathbf{E}(\neg a \mathbf{U} b) \wedge \neg \mathbf{E}(\neg a \mathbf{U}(a \wedge \mathbf{EXE}(\neg a \mathbf{U} b)))$$

Definition 2.24 (preceded_by-Semantik für $n..m$) Es gelte $p \models b \xrightarrow[n..m]{\text{preceded\_by}} a$ genau dann, wenn

$$p \models b \xrightarrow[0..m]{\text{preceded\_by}} a \;\wedge\; p \models b \xrightarrow[n..*]{\text{preceded\_by}} a$$

gilt.

succeeded_by-Semantik

Wie auch für preceded_by ist ein Merkmal der Semantikdefinitionen der succeeded_by-Bedingungsbeziehung, dass sie trivialerweise erfüllt sind, wenn keine Aktivität vom Quellaktivitätstyp im betreffenden Prozessinstanzmodell p auftritt.

Definition 2.25 (succeeded_by-Semantik für $0..m$) Es gelte $p \models b \xrightarrow[0..m]{\text{succeeded\_by}} a$ genau dann, wenn Folgendes gilt:

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \neg \mathbf{EF}(b \wedge \underbrace{\mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \ldots)}_{m+1 \text{ mal}} \ldots)$$

und

$$h(p) \in \{w \in AP'^* \mid \forall k \in \mathbb{N}_0 : w_k = b \Rightarrow |w_k...|_a \leq m\}$$

und

$$\forall k \in \mathbb{N}_0 \;:\; h(p)_k = b \Rightarrow \mathfrak{T}_p, \mathrm{pg}(p) \models \neg \underbrace{\mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \ldots)}_{m-|h(p)_k...|_a+1 \text{ mal}} \ldots).$$

Der definierende, rechtsseitige Ausdruck besteht aus drei Klauseln die wie folgt zu verstehen, sind:

1. Falls in der Prozesszukunft ein b durchgeführt wird, darf anschießend nicht $m + 1$ mal a ausgeführt werden.

2. Die zweite Klausel stellt sicher, dass die Prozesshistorie allein nicht der Bedingungsbeziehung widerspricht.

3. Die dritte Klausel in dem Ausdruck verbindet Prozesshistorie und -zukunft insoweit, dass, falls in der Prozesshistorie b durchgeführt wurde, in der

Prozesszukunft nicht $m - |h(p)_{k\ldots}|_a + 1$ mal a durchgeführt werden darf, also m-mal abzüglich der Vorkommen von a nach dem Vorkommen von b in der Prozesshistorie.

Für den Spezialfall $m = 2$ ist die CTL-Formel in der ersten Klausel:

$$\neg\mathbf{EF}(b \wedge \mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \mathbf{EXEF}a))).$$

Definition 2.26 (succeeded_by-Semantik für $n..*$) Es gelte $p \models b \xrightarrow[n..*]{succeeded\_by} a$ genau dann, wenn

$$\forall k \in \mathbb{N}_0 \;:\; h(p)_k = b \Rightarrow \mathfrak{T}_p, \mathrm{pg}(p) \models \underbrace{\mathbf{AXAF}(a \wedge \mathbf{AXAF}a \wedge \ldots}_{n-|h(p)_{k\ldots}|_a \text{ mal}}) \ldots)$$

und

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \mathbf{AG}(b \Rightarrow \underbrace{\mathbf{AXAF}(a \wedge \mathbf{AXAF}a \wedge \ldots}_{n-\text{mal}}) \ldots)$$

gilt.

Die erste Klausel im definierenden Ausdruck ist so zu verstehen, dass, falls b in der Historie durchgeführt wurde, a in der Prozesszukunft mindestens so häufig durchgeführt werden muss, wie n nach Abzug der Vorkommen von a in der Prozesshistorie vorschreibt. Die zweite Klausel besagt, dass, falls b in der Prozesszukunft durchgeführt wird, anschließend n-mal a durchgeführt werden muss.

Für den Spezialfall $m = 2$ ist die CTL-Formel in der letzten Klausel:

$$\mathbf{AG}(b \Rightarrow \mathbf{AXAF}(a \wedge \mathbf{AXAF}a))$$

Definition 2.27 (succeeded_by-Semantik für $n..m$) Es gelte $p \models b \xrightarrow[n..m]{succeeded\_by} a$ genau dann, wenn Folgendes gilt:

$$p \models b \xrightarrow[0..m]{succeeded\_by} a \;\wedge\; p \models b \xrightarrow[n..*]{succeeded\_by} a.$$

req_existence-Semantik

Die Semantik für req_existence-Ausdrücke wird zum Teil über LTL-Formeln definiert. Eine Definition in CTL ist nicht möglich. Im Wesentlichen sagen die Formeln aus, dass genau auf den Pfaden, auf denen b gilt auch mindestens n- bzw. maximal m-mal a gelten muss. In CTL ist es nicht möglich, komplexe Aussagen über Pfade zu treffen, da gemäß Syntaxdefinition (vgl. Definition 2.16), temporale Operatoren sich immer mit Pfadquantoren abwechseln müssen.

Auch für diese Bedingungsbeziehungen dieser Art gilt, dass sie trivialerweise gelten, wenn keine Aktivität vom Quellaktivitätstyp im betreffenden Prozessinstanzmodell vorhanden ist.

Definition 2.28 (req_existence-Semantik für $0..m$) Es gelte $p \models b \xrightarrow[0..m]{req\_existence} a$ genau dann, wenn

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \mathbf{A}(\mathbf{F}b \Rightarrow \neg \underbrace{\mathbf{XF}(a \wedge \mathbf{XF}(a \wedge \ldots \mathbf{XF}a))}_{m-|h(p)|_a+1 \text{ mal}} \ldots)$$

und

$$|h(p)|_b > 0 \Rightarrow \mathfrak{T}_p, \mathrm{pg}(p) \models \neg \underbrace{\mathbf{EXEF}(a \wedge \mathbf{EXEF}(a \wedge \ldots \mathbf{EXEF}(a}_{m-|h(p)|_a+1 \text{ mal}} \ldots)$$

und

$$|h(p)|_b > 0 \ \Rightarrow \ |h(p)|_a \leq m$$

gilt.

Die erste Klausel besagt, dass, falls in der Prozesszukunft ein b durchgeführt wird, in der Prozesszukunft nicht mehr als m-mal a durchgeführt werden darf, abzüglich der Vorkommen in der Prozesshistorie. Gleiches gilt, wenn b in der Prozesshistorie vorkommt (zweite Klausel). Die dritte Klausel fordert, dass, falls b in der Prozesshistorie vorkommt, maximal m-mal a in der Prozesshistorie vorkommen darf.

Definition 2.29 (req_existence-Semantik für $n..*$) Es gelte

$p \models b \xrightarrow[n..*]{req\_existence} a$ genau dann, wenn

$$\mathfrak{T}_p, \mathrm{pg}(p) \models \mathbf{A}(\mathbf{F}b \Rightarrow \underbrace{\mathbf{XF}(a \wedge \mathbf{XF}(a \wedge \ldots \mathbf{XF}a))}_{n-|h(p)|_a \text{ mal}} \ldots)$$

und

$$|h(p)|_b > 0 \Rightarrow \mathfrak{T}_p, \mathrm{pg}(p) \models \neg \underbrace{\mathbf{AXAF}(a \wedge \mathbf{AXAF}(a \wedge \ldots)}_{n-|h(p)|_a \text{ mal}} \ldots) \vee |h(p)|_a \geq n$$

gilt.

Die erste Klausel fordert, dass, falls in der Prozesszukunft ein b durchgeführt wird, in der Prozesszukunft mindestens n abzüglich der Vorkommen in der Prozesshistorie mal a durchgeführt werden darf. Falls b bereits in der Prozesshistorie vorkommt, müssen mindestens n-mal a in der Historie und der Zukunft vorkommen oder bereits mindestens n-mal in der Historie allein.

Definition 2.30 (req_existence-Semantik für *n..m*) Es gelte
$p \models b \xrightarrow[n..m]{req\_existence} a$ genau dann, wenn

$$p \models b \xrightarrow[0..m]{req\_existence} a \;\wedge\; p \models b \xrightarrow[n..*]{req\_existence} a$$

gilt.

Hiermit sind die Syntax- und Semantikdefinitionen der Prozessmodelle abgeschlossen. Offensichtlich ist, dass gerade die Bedeutungen von Bedingungsbeziehungen weitaus weniger trivial sind, als man zunächst vermuten könnte. Dies liegt nicht zuletzt daran, dass Bedingungsbeziehungen auf allgemeinen Prozessinstanzmodellen ausgewertet werden, die bereits im Normalfall eine Prozesshistorie beinhalten.

In Kapitel 4 werden die formalen Definitionen als Grundlage für die Realisierung einer werkzeuggestützten Komplianzprüfung von Prozessinstanzmodellen gegen Prozesswissensmodelle verwendet.

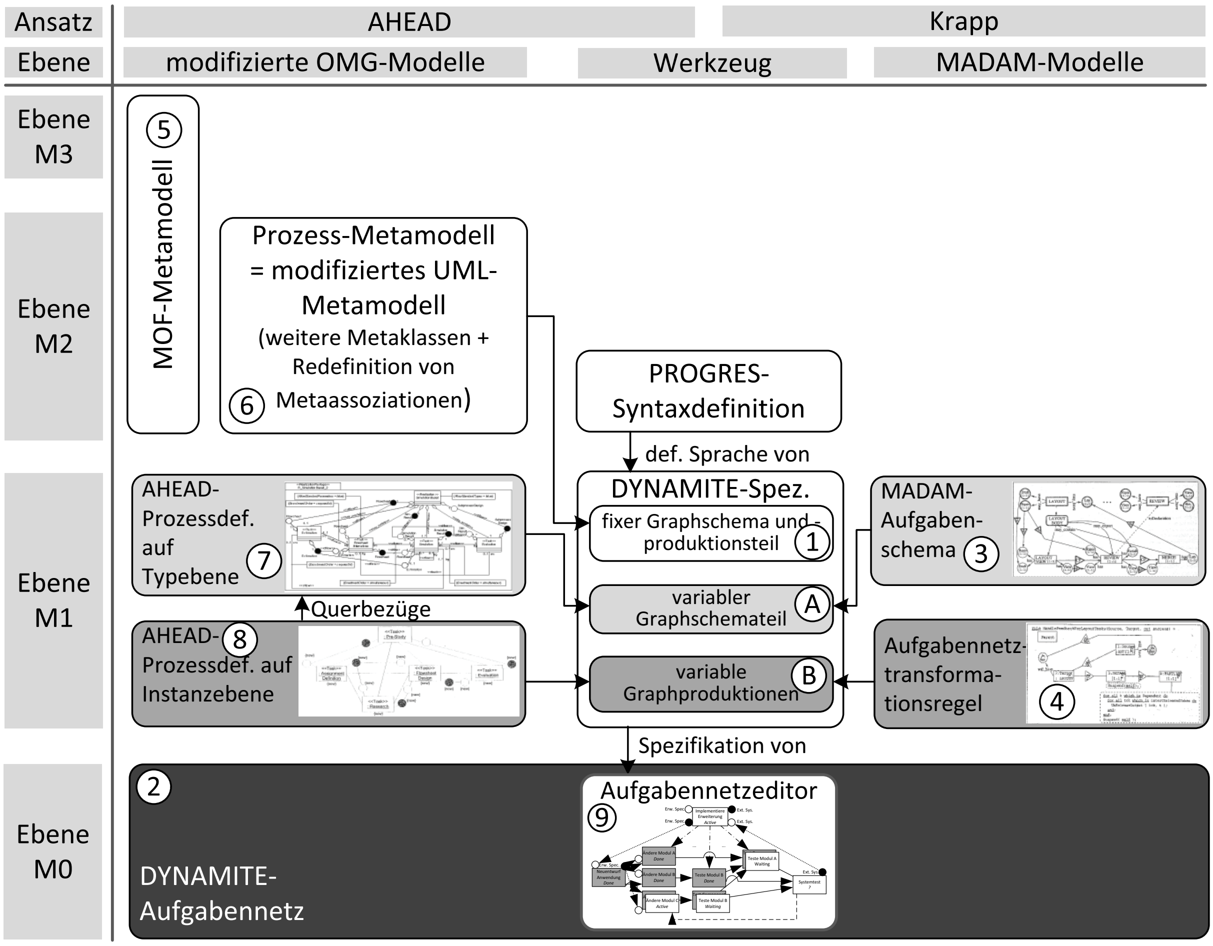


Abbildung 2.28: Modellschichtung im AHEAD-Ansatz

2.7 Vorarbeiten

Die Idee, Prozesse auf verschiedenen Abstraktionsebenen zu modellieren, und die Modelle einander in Beziehung zu setzen, ist nicht neu. Sie geht im Wesentlichen auf [NW94] zurück. Im Weiteren werden Vorgängerarbeiten beschrieben, in denen verschiedene Prozessmodellierungssprachen und Prozessmodellschichtungen entwickelt wurden. Die Prozessmodellierungssprachen und -schichtungen sind in Abbildung 2.28 grob wiedergegeben. Sie ähneln bei grober Betrachtung den zuvor beschriebenen, unterscheiden sich allerdings tatsächlich deutlich bei genauerer Betrachtung.

Die Prozessinstanzmodell-Sprache DYNAMITE

DYNAMITE [HJKW96, KKSW98, HKWJ97] steht für „DYNAMIc Task nEts“, zu Deutsch „Dynamische Aufgabennetze“, die für die Modellierung von Aktivitäten und deren Zusammenhängen in dynamischen Entwicklungsprozessen verwendet werden (vgl. Unterabschnitt 2.1.3). Zu DYNAMITE gehört ein for-

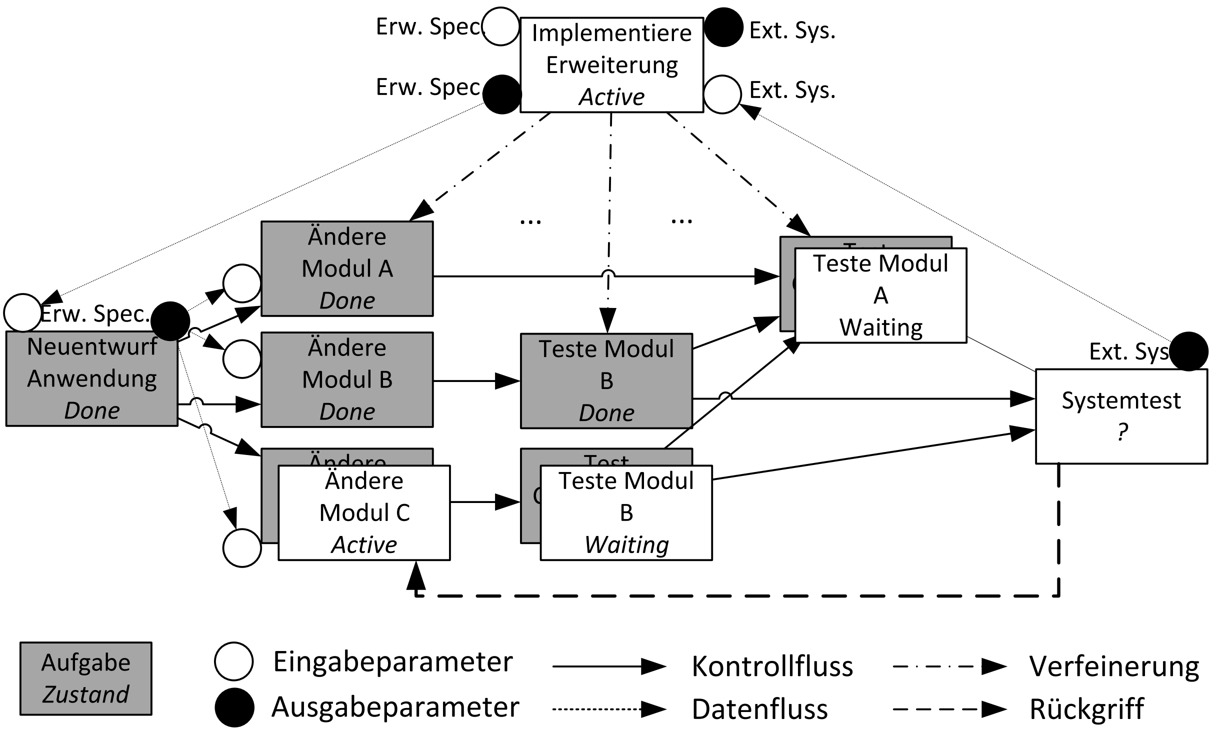


Abbildung 2.29: Beispiel eines Dynamischen Aufgabennetzes (nach [Sch02])

males, festgeschriebenes Graphschema (vgl. ① in Abbildung 2.28), das in PROGRES formalisiert ist. Dieses Graphschema spezifiziert einerseits die Syntax Dynamischer Aufgabennetze, also Sprachelemente und deren mögliche Kombinationen. Graphproduktionen bestimmen die Semantik, d.h. mögliche Änderungen an Dynamischen Aufgabennetzen.

Statische Struktur Dynamischer Aufgabennetze Abbildung 2.29 zeigt beispielhaft eine Dynamisches Aufgabennetz (vgl. ② in Abbildung 2.28). Das Beispiel ist aus Darstellungsgründen vereinfacht: Es ist nur ein kleiner Ausschnitt aus einem Softwareentwicklungsprozess abgebildet. Außerdem sind verschiedene Modellelemente, wie z.B. Datenflüsse, nur exemplarisch vorhanden und fehlen in dem Netz an Stellen, an denen sie sich realistischerweise befinden müssten.

Dynamische Aufgabennetze bestehen insbesondere aus *Aufgaben*. Aufgaben können *komplex* sein, d.h. selbst durch weitere Aufgaben weiter verfeinert werden oder *atomar*. Des Weiteren besteht jede Aufgabe immer aus einer *Schnittstelle*, die Informationen wie Ein- und Ausgabeparameter umfasst. Getrennt davon besitzt eine Aufgabe auch eine *Realisierung*. Im Fall einer komplexen Aufgabe ist die Realisierung die Menge verfeinernder Aufgaben, im Fall einer atomaren Aufgabe die Ressource, mit der die atomare Aufgabe letztlich durchgeführt wird.

Aufgaben können miteinander in Beziehung gesetzt werden. Eine bereits angesprochene Beziehung ist die *Verfeinerungsbeziehung* zwischen zwei Aufgaben, die zwischen einer komplexen Aufgabe und allen Aufgaben des verfeinernden Aufgabennetzes besteht. Da sich in der konkreten Darstellung Dynamischer Aufgabennetze eine komplexe Aufgabe im Allgemeinen über ihrem verfeinernden Subnetz befindet, wird die Verfeinerung als eine *vertikale Beziehung* klassifiziert.

Demgegenüber ist der *Kontrollfluss* eine *horizontale Beziehung*. Besteht zwischen zwei Aufgaben ein Kontrollflussbeziehung, so ist die mögliche Abarbeitungsreihenfolge zwischen diesen Aufgaben eingeschränkt. Diese Einschränkung hängt von der Art des Kontrollflusses ab. Generell gilt, dass eine Nachfolgeraufgabe erst dann endet, wenn die Vorgängeraufgabe beendet wurde. Bei *sequentiellen* Kontrollflüssen muss zusätzlich die Vorgängeraufgabe abgeschlossen sein, bevor die Nachfolgeraufgabe beginnen kann. Bei *simultanen* Kontrollflüssen kann eine Nachfolgeraufgabe erst dann starten, wenn die Vorgängeraufgabe gestartet wurde.

Eine weitere horizontale Beziehung ist der *Rückgriff* [KW97], der immer entgegengesetzt zu einem Kontrollfluss oder einem Pfad von Kontrollflüssen verläuft. Ein Rückgriff wird beispielsweise dann modelliert, wenn bei der Durchführung einer Aufgabe ein Fehler in einer bereits abgeschlossenen, bzgl. des Kontrollflusses vorgelagerten Aufgabe festgestellt wird. Ist die Zielaufgabe des Rückgriffs noch nicht abgeschlossen, so wird der betreffende Bearbeiter direkt über den Rückgriff informiert. Ist die Zielaufgabe bereits abgeschlossen, so wird die Aufgabe in neuer Version im Aufgabennetz angelegt und kann erneut gestartet werden.

Datenflüsse verlaufen immer von *Eingabeparametern* zu *Ausgabeparametern*. Hierüber wird der Datenaustausch zwischen zwei Aufgaben modelliert, die durch einen Kontrollfluss, einen Rückgriff oder eine Verfeinerungsbeziehung verbunden sind. Ausgetauschte Dokumente werden über *Tokens* realisiert, die Referenzen auf tatsächliche Versionen von Entwicklungsdokumenten darstellen.

Zustände in Dynamischen Aufgabennetzen Jede Aufgabe in einem Dynamischen Aufgabennetz befindet sich immer in einem von mehreren Zuständen. Standardmäßig gibt es pro Aufgabe sechs mögliche Zustände, deren mögliche Übergänge der Zustandsautomat in Abbildung 2.30 zeigt.

Die Zustände können nach Phasen gruppiert werden: In der Vorbereitung befindet sich die Aufgabe entweder im Anfangszustand InDefinition, in der die Aufgabe nur für den Prozessmanager aber nicht für Entwickler sichtbar ist, oder im Zustand Waiting, in der auch Entwickler die Aufgabe sehen können. Entwickler oder Manager können eine Aufgabe starten und somit

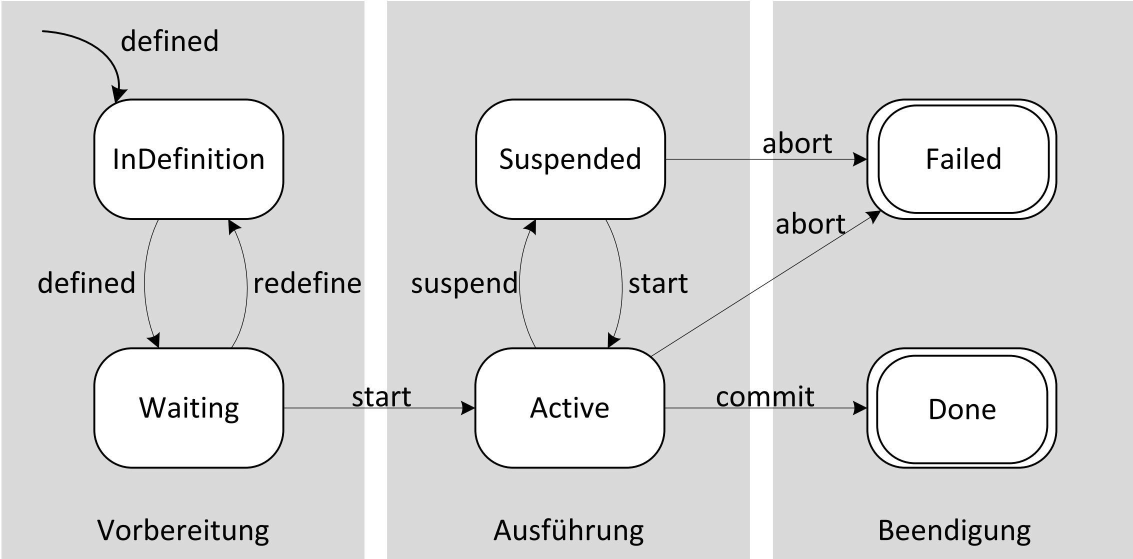


Abbildung 2.30: Zustandsautomat einer Aufgabe (nach [Sch02])

den Aufgabenzustand in die Phase Ausführung und den Zustand Active ändern. Soll angezeigt werden, dass die Ausführung einer Aufgabe gerade nicht sinnvoll fortgeführt werden kann, kann der Zustand zu Suspended gewechselt werden. Die Beendigung einer Aufgabe ist entweder erfolgreich, was durch den Zustand Done repräsentiert wird oder nicht erfolgreich, was den Wechsel zu Failed nach sich zieht.

Verhalten von Dynamischen Aufgabennetzen Nicht jede Kombination von Sprachelementen und Zuständen verbundener Aufgaben ergibt ein syntaktisch korrektes Aufgabennetz. DYNAMITE besitzt daher eine Menge von *strukturellen Invarianten*, die für jedes syntaktisch korrekte Aufgabennetz zu jedem Zeitpunkt gelten müssen. Zu den einfachen Regeln gehört, dass ein Datenfluss nur von einem Ausgabe- zu einem Eingabeparameter führen darf. Komplexere Regeln besagen, dass beispielsweise ein Rückgriff immer nur gegenläufig zu einem vorhandenen Kontrollfluss(-pfad) verlaufen darf.

Zustandsänderungen in Dynamischen Aufgabennetzen sind mit Bezug auf die Netzstruktur eingeschränkt. Der Produktautomat in Abbildung 2.31 gibt mögliche Zustandsübergänge von Zustandspaaren zweier durch eine Verfeinerungsbeziehung verbundener Aufgaben wieder. Man sieht hier insbesondere, dass viele (in der Abbildung weiß hinterlegte) Zustandspaare überhaupt nicht erreichbar sind, d.h. niemals eintreten. Beispielsweise kann sich eine verfeinernde Aufgabe nie im Zustand Done befinden, wenn die übergeordnete Aufgabe noch in der Vorbereitungsphase, also in InDefinition oder Waiting ist.

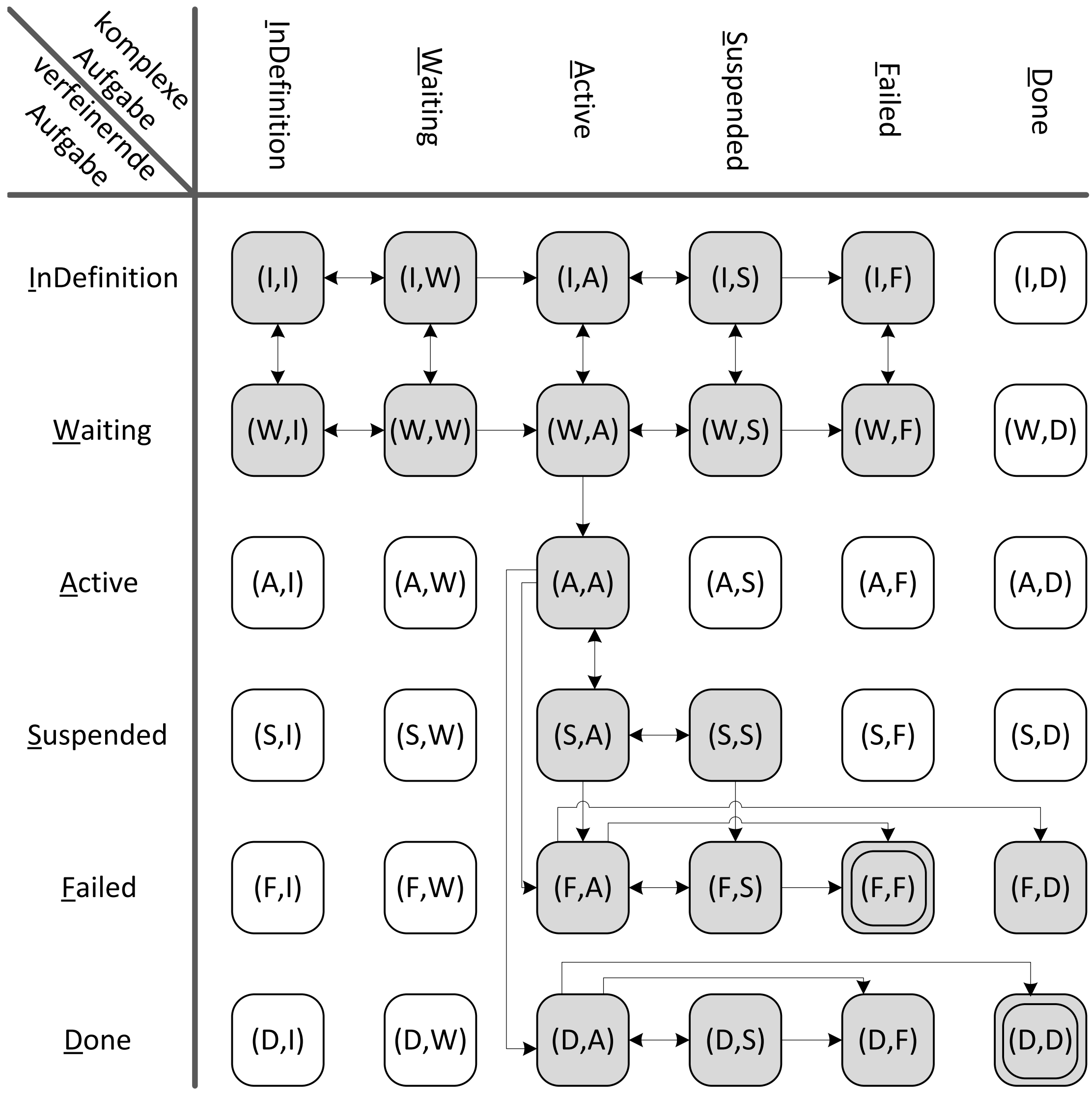


Abbildung 2.31: Zustandsproduktautomat zweier Aufgaben, die in einer Verfeinerungsbeziehung stehen (nach [Sch02])

Formale Spezifikation von DYNAMITE Ein wesentlicher Vorteil Dynamischer Aufgabennetze ist, dass ihre Editierung durch Werkzeuge informatisch unterstützt wird. Voraussetzung einer solchen Unterstützung ist jedoch, dass ein formales Metamodell existiert, das syntaktisch korrekte Dynamische Aufgabennetze und das Verhalten, also mögliche Änderungen an Dynamischen Aufgabennetzen formal spezifiziert. Im Fall von DYNAMITE geschieht dies über eine Spezifikation, die in PROGRES (PROgrammed Graph REwriting Systems) [Sch91] formuliert ist.

Die PROGRES-Spezifikation für DYNAMITE zerfällt in zwei wesentliche Bestandteile: Erstens enthält sie ein *Graphschema*, in dem Sprachbestandteile wie Aufgaben oder Kontrollflüsse eingeführt und miteinander in Beziehung gesetzt werden. Über das Graphschema können grundlegende syntaktische

Regeln festgehalten werden, beispielsweise, dass ein Datenfluss immer nur von einem Ausgabe- zu einem Eingabeparameter verlaufen darf. Zweitens enthält die Spezifikation eine Menge von *Graphproduktionen*, vergleichbar mit Graphersetzungsregeln in GROOVE-Graphgrammatiken (vgl. Unterabschnitt 2.4.2). Graphproduktionen ersetzen Teilstrukturen eines Dynamischen Aufgabennetzes durch andere Strukturen. Hierdurch können beispielsweise neue Aufgaben angelegt werden, bestehende Aufgaben durch Kontrollflüsse verbunden werden oder aber auch Zustände von Aufgaben verändert werden.

Das AHEAD-System

Eine PROGRES-Spezifikation kann entweder direkt interpretiert oder in C-Code überführt werden, der in das UPGRADE-Rahmenwerk [BJSW02] eingebettet werden kann. In einem so genannten UPGRADE-Prototypen kann ein Laufzeitgraph angezeigt werden und mittels der in der PROGRES-Spezifikation festgelegten Transaktionen manipuliert werden. Die Laufzeitgraphen werden dabei persistent in einer Graphdatenbank gespeichert, beispielsweise GRAS [Bau99, KSW95] oder DRAGOS [Böh06].

Das AHEAD-System [JSBW00, Wes01, HJK^{+}08] beinhaltet einen Aufgabennetzeditor, der ein UPGRADE-Prototyp ist. Der Aufgabennetzeditor speichert ein Aufgabennetz intern als getypten, attributierten und gerichteten Graphen, der den syntaktischen Bedingungen des Graphschemas genügt und der nur mittels bestimmter Graphproduktionen und Graphtransaktionen verändert werden kann. Ein Aufgabennetz sieht aus wie in Abbildung 2.29 gezeigt. Hierbei werden gegenüber der internen Graphdatenstruktur gewisse Vereinfachungen vorgenommen, z.B. werden Kanten zwischen Aufgaben und zugehörigen Parametern nicht durch Pfeile sondern durch aneinander grenzende Anordnung dargestellt.

Die Funktionalität von AHEAD geht über die der reinen Darstellung und Manipulation von Aufgabennetzen hinaus. AHEAD verwaltet neben dynamischen Aufgabennetzen auch Modelle für das Management von Ressourcen und Entwicklungsprodukten (Entwicklungsdokumente), sowie Querbezüge zwischen Aufgaben, Ressourcen und Produkten. In AHEAD können neben der für Prozessmanager geeigneten Sicht auf dynamische Aufgabennetze, wie in Abbildung 2.29 dargestellt, auch dedizierte Sichten für Entwickler angezeigt werden. In diesen Entwicklersichten werden nur die für die Bearbeitung einer bestimmten Aufgabe relevanten Informationen graphisch angezeigt.

Neben diesen rollenspezifischen Sichten bietet AHEAD konfigurierbare Sichten für übergreifende Prozesse, die von mehreren kooperierenden Parteien durchgeführt werden. Hierdurch können private Teile eines Aufgabennetzes bei den jeweiligen Kooperationspartnern ausgeblendet werden

[HW06a, HW06b, HW07, Hel08].

Modellierung und Auswertung von Prozesswissen Die Spezifikation der AHEAD zugrunde liegenden Teilmodelle, insbesondere DYNAMITE, ist so allgemein gehalten, dass sie für beliebige Entwicklungsprozesse geeignet ist. Unterschiedliche Arten von Entwicklungsprozessen sowie das Prozesswissen in unterschiedlichen Firmen unterscheiden sich voneinander. Deshalb ist die Unterstützung einer geeigneten Modellierung von Prozesswissen nützlich. Hierbei sollte das modellierte Prozesswissen beim Management konkreter Entwicklungsprozesse mittels der bereits beschriebenen Werkzeuge genutzt werden.

Im Folgenden werden zwei Arbeiten beschrieben, die sich mit der Adaption Dynamischer Aufgabennetze an formalisiertes Prozesswissen beschäftigen [Kra98] bzw. die Formalisierung und Evolution des Prozesswissens auf die verbreitete Modellierungssprache UML gründen [Sch02].

Adaption dynamischer Aufgabennetze Krapp beschreibt in [Kra98], wie die Basisspezifikation von DYNAMITE um domänen- oder firmenspezifisches Prozesswissen angereichert werden kann. Dies geschieht über die eigens entwickelte Modellierungssprache MADAM. Der Zweck von MADAM ist, dem Modellierer eine, im Vergleich zu PROGRES einfach zu verwendende Sprache zur Verfügung zu stellen, über die er Prozesswissen so formalisieren kann, dass es später bei der Editierung von Dynamischen Aufgabennetzen verwendet werden kann. Die Formalisierung geschieht über verschiedene, graphische und textuelle Modelle.

Aufgabenschemata Über Aufgabenschemata (vgl. ③ in Abbildung 2.28) können zusätzlich zu den generischen Modellelementen der DYNAMITE-Basisspezifikation weitere spezifische Modellelemente deklariert werden, die die generischen Elemente verfeinern. So können in einem Entwicklungsprozess für ein mechanisches Bauteil in Aufgabennetzen statt allgemeinen Aufgaben vom Typ `Task` spezifische Aufgaben, beispielsweise vom Typ `Layout`, verwendet werden.

Auch ohne Aufgabenschemata kann für den menschlichen Betrachter der fachliche Typ einer Aufgabe durch geeignete Benennung dargelegt werden, beispielsweise durch die Benennung `Layout Housing`. Jedoch ermöglichen Aufgabenschemata, Bezüge zwischen Aufgabentypen zu formulieren und bei der Editierung von Dynamischen Aufgabennetzen zu verwenden. Abbildung 2.32 zeigt beispielhaft, wie eine Aufgabe vom Typ `Layout` durch ein bestimmtes Subnetz verfeinert werden kann, in dem sich wiederum Aufgaben vom Typ `Layout View`, `Review` und `Merge` befinden können. Zwischen diesen

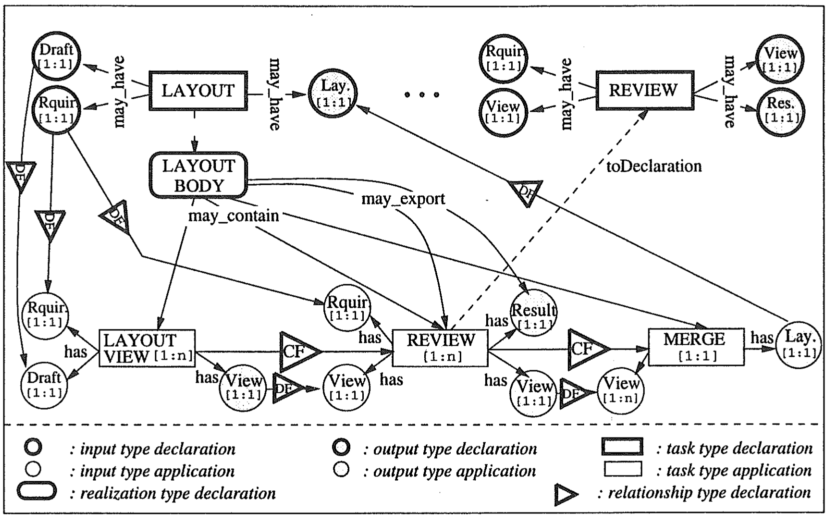


Abbildung 2.32: Beispiel eines Aufgabenschemas [Kra98]

Aufgaben im Subnetz können des Weiteren Kontroll- und Datenflussabhängigkeiten formuliert werden. Diese Abhängigkeiten erzwingen sinnvollerweise, dass in den betreffenden Subnetzen in einem Dynamischen Aufgabennetz Aufgaben vom Typ `Layout View` stets vor Aufgaben vom Typ `Review` durchgeführt werden und diese vor einer gemeinsamen `Merge`-Aufgabe [Kra98, Seite 134].

So wie die DYNAMITE-Basisspezifikation nur generische Modellelemente und deren Verwendung definiert, sind auch die Operationen zur Manipulation von Aufgabennetzen generisch und elementar. Über Aufgabennetz-Transformationsregeln (vgl. ④ in Abbildung 2.28) können spezifische und komplexe Editieroperationen spezifiziert werden, beispielsweise die Behandlung eines Rückgriffs speziell für `Layout`-Aufgaben wie in Abbildung 2.33 dargestellt [Kra98, Seite 141].

Neben Aufgabenschemata und Aufgabennetz-Transformationsregeln lassen sich in MADAM auch aufgabenbezogene Ereignisse und deren Behandlung über so genannte *Event-Trigger-Rules* modellieren. Aufgabenschemata lassen sich darüberhinaus über eine textuelle Modellierungssprache um weitere *Bedingungen* anreichern. Des Weiteren unterstützt MADAM die Redefinition von Operationen des Basismodells. So kann die allgemeine Operation zum Start einer allgemeinen Aufgabe in einer spezifischen Operation zum Start von Aufgaben vom Typ `PartList` redefiniert werden [Kra98, Seite 138].

Die Semantik von MADAM ist durch Transformation in eine PROGRES-Spezifikation formal erklärt. Das Ergebnis der Transformation reichert dabei die Basis-Spezifikation von DYNAMITE an [Wes99b, Seite 256]. Die Editierunterstützung für Modelle in MADAM ist selbst wiederum in großen Teilen über einen mittels PROGRES spezifizierten Prototypen realisiert.

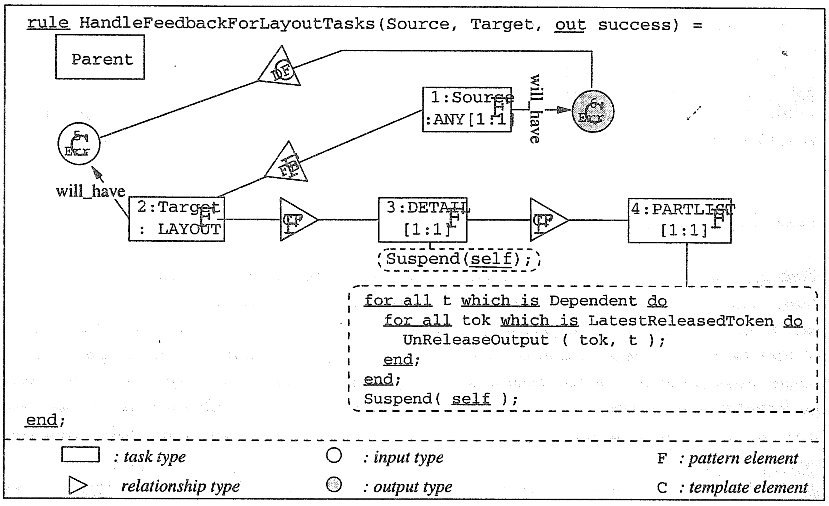


Abbildung 2.33: Beispiel einer Aufgabennetz-Transformationsregel [Kra98]

Evolution und UML-basierte Formalisierung von Prozesswissen Die Modellierungssprache MADAM verfügt über eine eigens entwickelte Notation, die von einem Prozessmodellierer zunächst zu erlernen ist. Schleicher et. al beschreiben in [HSW03, Sch02] einen alternativen Ansatz, der Aufgabenschemata und Transformationsregeln in die Unified Modeling Language (UML) einbettet. Prozessmodelle können damit als Varianten von UML-Kollaborations- und -Klassendiagrammen verfasst werden. Nach Ansicht von Schleicher et al. besteht der Vorteil dieses Ansatzes darin, dass, aufgrund ihrer Verbreitung, Kenntnisse über UML bei vielen Prozessmodellierern vorausgesetzt werden können.

Der UML-basierte Ansatz orientiert sich an der aus vier Ebenen bestehenden Metadaten-Architektur der Meta Object Facility (MOF) der Object Management Group (OMG) wie in Unterabschnitt 2.1.4 beschrieben. Die Abbildung 2.28 zeigt die in [Sch02] entwickelte Modellschichtung.

Auf der obersten, *dritten* Ebene befindet sich das Prozess-Metametamodell. Dieses ist das MOF-Metamodell ⑤. Hiervon ist auf der *zweiten* Ebene das Prozess-Metamodell ⑥ eine Modellinstanz. Prozessmodelle auf der *ersten* Ebene sind wiederum Modellinstanzen des Prozess-Metamodells. Sie modellieren Aufgabennetze auf nullter Ebene.

Da dem Prozessmodellierer eine UML-ähnliche Modellierungssprache bereitgestellt werden soll, ist das Prozess-Metamodell nicht von Grund auf neu entworfen sondern ist eine Variante des UML-Metamodells. Hierbei wird das UML-Metamodell um weitere Metaklassen und -assoziationen erweitert [SW01]. Diese Erweiterungen bewirken zweierlei: Zum Einen wird sichergestellt, dass gültige Instanzen des Metamodells auch nur sinnvolle Prozessmodelle sind. Zum Anderen werden erst durch die Erweiterungen Prozessmodelle maschineninterpretierbar, d.h. insbesondere, dass erst durch sie automatische Prüfungen der Modellkonsistenz möglich sind. Es wird also durch die Änderungen des Metamodells die Syntax der diagrammatischen Sprache UML auf die Erfordernisse der Modellierung von Prozessdefinitionsmodellen angepasst.

Prozesse können in AHEAD-Prozessdefinitionsmodellen abstrakt modelliert werden. Je nachdem welche Genauigkeit erreicht werden soll, können unterschiedliche Modellierungssprachen verwendet werden. Ist für bestimmte Prozesse bzw. Prozessteile das rein typologische Vorgehen klar, d.h. Art und Reihenfolge von Aufgaben nicht aber deren Anzahl, so kann dieses in *AHEAD-Prozessdefinitionen auf Typebene* ⑦ festgehalten werden. Diese Art von AHEAD-Prozessdefinitionen auf Typebene ähnelt syntaktisch den bekannten UML-Klassendiagrammen, weißt aber Modellelemente wie Aufgaben, Ein- und Ausgabeparameter auf, die erst durch die angesprochene Erweiterung des UML-Metamodells eingeführt wurden. Hinsichtlich des Abstraktionsniveaus sind AHEAD-Prozessdefinitionen auf Typebene mit den MADAM-Aufgabenschemata vergleichbar.

Andere Prozessteile, deren Struktur bekannt ist, können in spezielleren, dafür präziseren *AHEAD-Prozessdefinitionen auf Instanzebene* ⑦ formuliert werden, die syntaktisch UML-Kollaborationsdiagrammen ähneln. Das Analogon in MADAM sind die Aufgabennetz-Transformationsregeln. Im Gegensatz zu Prozessdefinitionen auf Typebene wird auf Instanzebene nicht von Kardinalitäten von Aufgaben eines bestimmten Typs abstrahiert. Prozessdefinitionen auf Instanzebene sehen zudem vor, Modellelemente mit zusätzlichen Markierungen zu versehen, über die erlaubte, komplexe Änderungen an Dynamischen Aufgabennetzen modelliert werden können.

Die Prozessdefinitionen können im UML-Editor Rational Rose modelliert werden. Hierbei können Querbezüge zwischen Prozessdefinitionen auf Instanz und auf Typebene ausgenutzt werden. Modellelemente in AHEAD-Prozessdefinitionen auf Instanzebene beziehen sich auf AHEAD-Prozessdefinitionen auf Typeebene. Die Querbezüge sind dabei ähnlich zu denen aus Abbildung 2.7. Widersprüche zwischen Modellen beider Arten werden von Rational Rose aufgedeckt.

Abbildung 2.28 zeigt den Zusammenhang zwischen dem Prozessmetamodell, den in Rational Rose modellierten AHEAD-Prozessdefinitionen und dem Aufgabennetzeditor zur Editierung von DYNAMITE-Aufgabennetzen. Der Aufgabennetzeditor (UPGRADE-Prototyp) ist dabei ebenfalls in die 4-Ebenen-Modellschichtung eingeordnet, wobei die Beziehung zwischen den Modellebenen hier nicht mit einer Modellinstanzbeziehung erklärt werden kann: Die DYNAMITE-Spezifikation ist in der formalen Spezifikationssprache PROGRES verfasst. Deren Syntax wird in [Sch91] formal definiert. Die Syntax-Definition von PROGRES ist auf Ebene M2 anzusiedeln, und die DYNAMITE-Spezifikation auf Ebene M1, da sie in PROGRES verfasst ist. DYNAMITE-Aufgabennetze stellen die manipulierbaren Graphen dar. Diese Graphen werden nach Schleicher [Sch02] als die zu modellierenden Informationen auf M0-Ebene aufgefasst.

Diese Zuordnung unterscheidet sich von der dieser Arbeit. Wie in Abbildung 2.12 gezeigt, sind Modelle aller drei Arten in dieser Arbeit der M1-Ebene zugeordnet. Prozessinstanzmodelle sind daher keine Modellinstanzen von Prozessdefinitions- oder Prozesswissensmodellen im Gegensatz zu AHEAD-Aufgabennetzen, die als Instanzen von AHEAD-Prozessdefinitionen (auf Typ- und Instanzebene) aufgefasst werden. Der Unterschied beider Ansätze resultiert daraus, dass unterschiedliche Werkzeuge und Metamodellierungsrahmenwerke eingesetzt wurden. In dieser Arbeit wird die Werkzeugunterstützung auf Basis verschränkter Ecore-Metamodelle realisiert. Gerade die Verschränkung wird bei der Realisierung bestimmter Prüffunktionalitäten ausgenutzt (vgl. Kapitel 4). Bei AHEAD wird der UML-Modellierungsansatz mit PROGRES-Spezifikationen kombiniert. Eine Verschränkung der Metamodelle im Sinne von Abbildung 2.14 ist nur zwischen Prozessdefinitionen auf Instanzebene und Prozessdefinitionen auf Typebene gegeben. Die Verschränkung besteht hier gerade aus dem in Abbildung 2.7 erläuterten Querbezug zwischen UML-Objekt und -Klasse. AHEAD-Aufgabennetze hingegen sind keine UML-Modelle. Umfassende Prüffunktionen für AHEAD-Aufgabennetze wurden daher nicht unter Ausnutzung von Verschränkung sondern auf anderem Wege erreicht [Sch02].

Die DYNAMITE-Spezifikation ist zweigeteilt in einen fixen Teil Ⓐ und einen variablen Teil Ⓑ. Das Prozess-Metamodell wird als unveränderlich angenommen. Daher finden sich die Festlegungen des Prozess-Metamodells im fixen Graphschema ① der Spezifikation wieder. Dieses Graphschema wird erweitert durch Anteile, die aus den AHEAD-Prozessdefinitionen auf Typebene automatisch generiert werden können. Die möglichen Manipulationen von Aufgabennetzen werden durch Graphproduktionen festgelegt. Komplexe Graphproduktionen können dabei aus Prozessdefinitionen auf Instanzebene automatisch gewonnen werden. Die Prozessdefinitionen wirken also als Parametrisierung einer vorhandenen Basisspezifikation von DYNAMITE.

Abgrenzung

Trotz der Tatsache, dass sich die unterschiedlichen Prozessmodellarten und Metamodelle der Vorgängerarbeiten auf ähnliche Weise in eine Modellschichtung einordnen lassen, unterscheidet sich der Ansatz dieser Arbeit deutlich von den Vorgängern.

Prozessinstanzmodelle Die Sprache DYNAMITE wurde von Grund auf neu entwickelt. Ziel des Sprachentwurfs von DYNAMITE war von Anfang an, hochdynamische und komplexe Entwicklungsprozesse angemessen modellieren zu können. DYNAMITE unterscheidet sich von SimBPEL-Instance daher stark. Gemeinsamkeiten bestehen darin, dass auch DYNAMITE graphisch notiert werden kann, mit Aktivitäten (Aufgaben) als Knoten und Ausführungspräzedenzen als Kanten. DYNAMITE bietet darüber hinaus die Möglichkeit, Datenflüsse zwischen Aktivitäten feingranular zu modellieren. Zusammen mit den komplementären Sprachen ResMod und Coma lassen sich alle für das Management von Entwicklungsprozessen wichtigen Aspekte modellieren. DYNAMITE ermöglicht jedoch gegenüber SimBPEL(-Instance) weniger Flexibilität zur Modellierungszeit. Zur Kontrollflussdefinition stehen nur einfache Kontrollflusskanten zur Verfügung. Schleifen und Verzweigungen stehen nicht zur Verfügung. Demzufolge bedeutet ein einfaches Vorkommen einer Aktivität in einem DYNAMITE-Aufgabennetz, dass diese Aktivität genau einmal durchgeführt wird bzw. worden ist. Die Durchführung einer Aktivität in einem WS-BPEL-Prozess ist immer abhängig davon, ob sie innerhalb einer Verzweigung (`Case`-Element) oder Schleife (`While`-Element) modelliert wird. Die geringere Flexibilität zur Modellierungszeit von DYNAMITE ist bewusst gewählt, da Entwicklungsprozesse nie a-priori vollständig modelliert werden können und daher die Modellierungs- und Ausführungszeit von Aufgabennetzen stets verschränkt ist. Wie in Kapitel 3 erläutert wird, ist die Bereitstellung von Funktionalitäten für dynamische Änderungen von in WS-BPEL modellierten Prozessen aufgrund der erhöhten Flexibilität zur Modellierungszeit mit zusätzlichen, konzeptuellen Schwierigkeiten verbunden.

Prozessdefinitionsmodelle Der Abstraktionsschritt zwischen DYNAMITE-Aufgabennetzen und AHEAD-Prozessdefinitionen auf Instanzebene ist ähnlich zu dem zwischen SimBPEL(-Instance) und SimBPEL. Im Wesentlichen wird in den Prozessdefinitionsmodellen von konkreten Ausführungszuständen abstrahiert. Der besagte Unterschied bzgl. Modellierungszeitflexibilität bleibt aber auch hier erhalten: AHEAD-Prozessdefinition auf Instanzebene beinhalten keine Verzweigungen und Schleifen.

Prozesswissensmodelle Auch die Sprachen für Prozesswissensmodelle unterscheiden sich in beiden Ansätzen. Dies ist eine Folge des Unterschieds zwischen AHEAD-Prozessdefinitionen auf Instanzebene und DYNAMITE-Aufgabennetzen auf der einen Seite und Prozessdefinitions- und -instanzmodellen auf der anderen Seite. Eine AHEAD-Prozessdefinition auf Typebene muss sprachlich so gestaltet sein, dass AHEAD-Prozessdefinitionen auf Instanzebene und DYNAMITE-Aufgabennetze gegen sie auf Komplianz geprüft werden

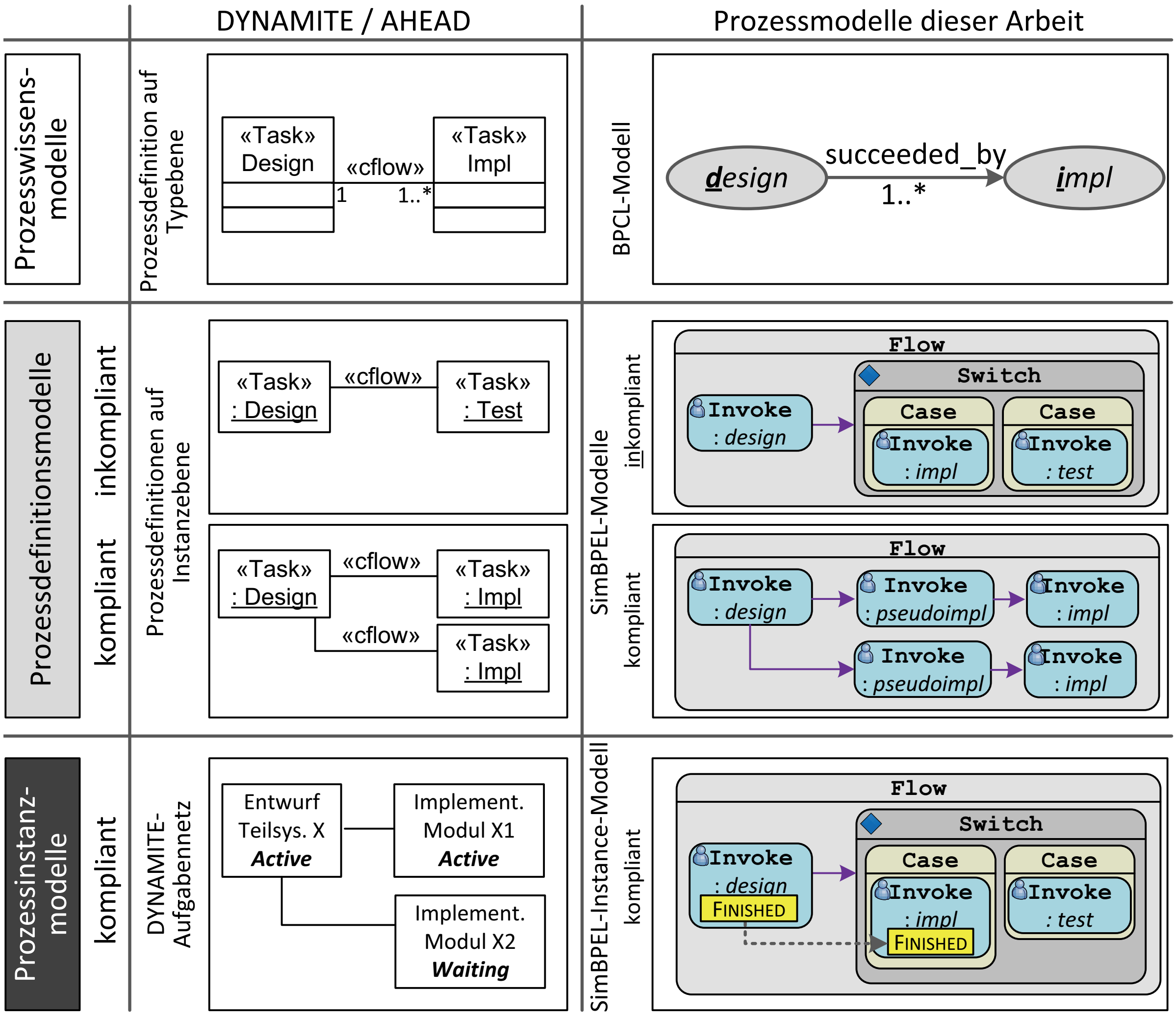


Abbildung 2.34: Prozessmodelle im Vergleich

können. Genauso verhält es sich zwischen BPCL-Modellen und SimBPEL- bzw. SimBPEL-Instance-Modellen (vgl. Kapitel 4).

Beispiel 2.16 (Komplianz im Vergleich) In Abbildung 2.34 sind einfache Prozessmodelle beider Ansätze einander gegenübergestellt. Trotz ihrer Ähnlichkeit besagen die Prozesswissensmodelle Unterschiedliches: Die AHEAD-Prozessdefinition auf Typebene macht direkt eine Aussage über die Struktur der Prozessdefinitionen auf Instanzebene und DYNAMITE-Aufgabennetze. Eine Design-Aktivität muss über einen Kontrollfluss mit mindestens einer Impl-Aktivität verbunden sein. Fehlt für eine Design-Aktivität eine adjazente Impl-Aktivität, so ist das Prozessdefinitions- (AHEAD-Prozessdefinition auf Instanzebene) bzw. -instanzmodell (DYNAMITE-Aufgabennetz) nicht kompliant in Bezug auf die AHEAD-Prozessdefinition auf Typebene. Im Falle von BPCL ist es so, dass ausgedrückt wird, dass in jeder Prozessdurchführung für jede *design*-Aktivität mindestens eine *impl*-Aktivität irgendwann danach ausgeführt werden muss. Dementsprechend ist das obere SimBPEL-Modell nicht kompliant, da die *impl*-Aktivität nicht in jedem Fall durchgeführt wird. Das unter SimBPEL-Modell ist hingegen kompliant, obwohl die *impl*-Aktivitäten in der Modellstruktur nicht adjazent zu der *design*-Aktivität sind. BPCL-Modelle treffen also nur indirekt Aussagen über SimBPEL(-Instance)-Modelle. Stattdessen beziehen sie sich direkt auf das Transitionssystem, das aus einem SimBPEL(-Instance)-Modell generiert werden kann.

Ein weiterer Effekt ist, dass SimBPEL-Instance-Modelle, die zunächst eine Komplianzbedingung verletzen, allein durch das Fortschreiten des Kontrollflusses die Komplianzbedingung zu einem bestimmten Zeitpunkt erfüllen. Dieser Zeitpunkt ist beispielsweise in dem SimBPEL-Instance-Modell aus Abbildung 2.34 erreicht, zu dem feststeht, dass die *impl*-Aktivität tatsächlich durchgeführt wird.

Kapitel 3

Erweiterung statischer Prozessmanagementsysteme

Wie in Unterabschnitt 1.3.2 motiviert, sind auch in der Versicherungsbranche Prozesse dynamisch und können nicht vollständig im Vorhinein geplant werden. In Abschnitt 1.2 wurden die Probleme bei der Unterstützung dynamischen Prozesse durch klassische, d.h. statische, Prozessmanagementsysteme diskutiert. Ergebnis der Diskussion war, dass naheliegende Lösungsansätze wie beispielsweise die Verwendung von Maximalmodellen unzureichend sind. Stattdessen ist bei Geschäftsprozessen eine Mischform richtig: Gängige Abläufe werden unter Verwendung flexibler Modellierungskonstrukte wie Verzweigungen oder Schleifen vorab modelliert, unvorhergesehene und seltene Abläufe durch dynamische Änderungen im jeweiligen Prozessinstanzmodell zur Prozesslaufzeit umgesetzt.

Gängige Prozessmanagementsysteme sind allerdings so implementiert, dass Änderungen in der Ablaufstruktur in Prozessinstanzmodellen nicht unterstützt werden. Vergleichende Studien hierzu finden sich in [WRRM08, WRR07a, WRR07b] sowie [Rei00, Kapitel 10].

In diesem Kapitel wird, nach der Einordnung verschiedener Prozessmanagementsysteme hinsichtlich ihrer Eignung für dynamische Prozesse (Abschnitt 3.1) ein Ansatz erläutert, über den das statische Prozessmanagementsystem WebSphere Process Server so erweitert werden kann, dass dynamische Änderungen in Prozessinstanzmodellen möglich sind. Abschnitt 3.2 umreißt den dabei verfolgten Simulationsansatz und die daraus resultierende Systemarchitektur, Abschnitte 3.3 und 3.4 detaillieren den Ansatz in Hinblick auf unterstützte Änderungsmuster anhand einer GROOVE-Graphgrammatik. Die Graphgrammatik dient nur der konzeptuellen Exploration und der Darstellbarkeit der Simulationskonzepte. In Abschnitte 3.5 und 3.6 wird daher dargelegt, wie sich diese Konzepte in das eigentliche technische Umfeld des WebSphere Process Servers übertragen lassen. Abschnitte 3.7 und 3.8 schließen das Kapitel mit einer Diskussion und einem Vergleich mit verwandten Arbeiten.

3.1 Prozessmanagementsysteme

Es herrscht in der Literatur keine einheitliche Auffassung, welche exakten Eigenschaften ein Prozessmanagementsystem ausmachen. Der bestehende Konsens wird in den folgenden Unterabschnitten dargestellt, wobei zunächst Prozessmanagementsysteme nach außen von anderen Systemklassen abgegrenzt werden. In der anschließenden inneren Abgrenzung werden verschiedene Arten von Prozessmanagementsystemen insbesondere hinblicklich ihrer Eignung bzgl. Unterstützung dynamischer Prozesse diskutiert.

3.1.1 Abgrenzung nach außen

Nach [HHR04] ist ein Informationssystem ein System zur Information und Kommunikation. Prozessmanagementsysteme werden gemeinhin als Unterklasse von Informationssystemen aufgefasst mit einer bestimmten Zweckbestimmung. Bei Prozessmanagementsystemen werden Informationen über Prozesse verwaltet und zwischen verschiedenen Systemen bzw. zwischen System und menschlichen Verwendern kommuniziert.

Als Informationssysteme weisen Prozessmanagementsysteme gegenüber anderen Systemklassen typische *Merkmale* auf:

Prozessorientierung ist die dominante Eigenschaft von Prozessmanagementsystemen. Sie besagt, dass Eingaben und Ausgaben dieser Systeme sich immer auf Prozesse beziehen. Typische Eingaben und Ausgaben für Prozessmanagementsysteme sind beispielsweise Zustandsänderungen einzelner Aktivitäten. Andere Eingaben können Prozessdefinitionsmodelle sein, also Modelle von Prozesstypen.

Verteiltheit von Prozessmanagementsystemen kann unterschiedlich stark ausgeprägt sein. Normalerweise sind zumindest Systemkern und Benutzungsschnittstellen örtlich entfernt voneinander, d.h. laufen in unterschiedlichen Betriebssystemprozessen auf unterschiedlicher Hardware. Bei abteilungs- oder organisationsübergreifenden Prozessen werden aus Effizienzgründen und Gründen der Vertraulichkeit interner Daten häufig auch mehrere verteilte Systemkerne betrieben, um einen übergreifenden Prozess zu unterstützen. Für Entwicklungsprozesse wurde dies beispielsweise in [Hel08] und [Jäg03] studiert.

Reaktivität ist Prozessmanagementsystemen insofern zuzuschreiben, als dass sie normalerweise durchgängig im Betrieb sind und keine Funktion, wie beispielsweise Übersetzer berechnen, sondern reaktiv auf Eingaben durch Zustandsänderungen (z.B. Änderungen der Fortschrittsinformationen in

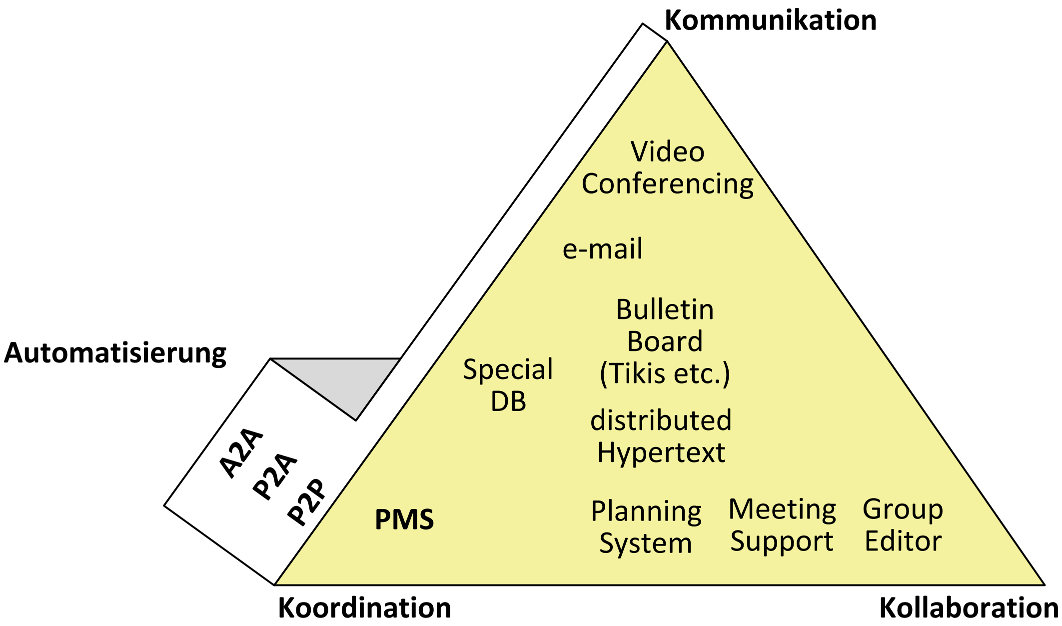


Abbildung 3.1: Zwecke von Prozessmanagementsystemen (nach [TSMB95])

einer Prozessinstanz) reagieren, die an Prozessbeteiligte oder verbundene Systeme kommuniziert werden.

Parametrisierbarkeit ist für Prozessmanagementsysteme insoweit wichtig, als dass ihr Einsatzumfeld stark variieren kann. Zwingend notwendig ist für solche Systeme in jedem Fall, dass sich die im jeweiligen Einsatzumfeld relevanten Prozesse präzise in Modelle abbilden lassen, die vom Prozessmanagementsystem interpretiert werden können.

3.1.2 Abgrenzung nach innen

Der Zweck von Prozessmangagementsystemen ist hauptsächlich der der Koordination von Arbeitsprozessen. Trotzdem dienen solche Systeme – wenn auch in geringerem Maße – anderen Zwecken. In Abbildung 3.1 findet sich hierzu eine Einordnung. Hierbei ist unter Koordination sowohl die Koordination von technischen als auch menschlichen Ressourcen zu verstehen, die in einem Prozess eingebunden sind.

Die in Abbildung 3.1 eingeordnete Systeme unterstützen in erster Linie solche Prozesse, die von Menschen durchgeführt werden. Im Bereich der Prozessmanagementsysteme ist eine weitere Unterteilung nach [DAH05] sinnvoll, anhand derer Prozessmanagementsysteme untereinander unterschieden werden können.

People-to-People (P2P) Systeme werden hauptsächlich zur Unterstützung von gänzlich manuell durchgeführter Prozesse eingesetzt. Zu diesen Systemen gehören insbesondere solche, in denen Prozesse nicht explizit modelliert werden und die einen Prozess daher nicht koordinierend unterstützen. Hierzu zählen *Groupware*-Systeme, auch *Computer-Supported-Collaborative-Work-Systeme (CSCW)* genannt. Diese Systeme stellen in erster Linie Funktionen zur Ablage und kollaborativen Bearbeitung gemeinsamer Daten bereit. Diese Daten sind meistens Dateien aus gängigen Anwendungen wie Textverarbeitungen oder Tabellenkalkulationen. *Content Management Systeme (CMS)* hingegen verwalten Inhalte ähnlich wie Groupware-Systeme, jedoch mit der Absicht, sie großen Leserschaften (Firmenmitarbeiter) oder weltweit zugänglich zu machen. Projektmanagementsysteme modellieren Prozesse explizit. In diesen Modellen ist der Aktivitäts- und Ressourcenaspekt dominierend, Daten- und Datenflüsse werden nicht modelliert. Hierdurch eigenen sich solche Systeme für die Planung komplexer Entwicklungsprozesse, insbesondere auch deswegen, weil Termin- und Kostenprobleme durch die Modellierbarkeit von Durchführungsdauern und Ressourcenverfügbarkeiten erkannt werden können. In beschränktem Maße können Projektmanagementsysteme auch Prozesse koordinieren. Hierbei ist die Koordination aber auf rein menschliche Ressourcen beschränkt.

People-to-Application (P2A) Systeme sind häufig Teile größerer Systeme wie Enterprise-Resource-Planning- (ERP), Customer-Relationship-Management- (CRM) oder Supplier-Relationship-Management-Systeme (SRM). Ein typisches Merkmal der hierbei verwendeten Prozessmodelle ist, dass sie die Dialogsteuerung zwischen häufig nur einem menschlichen Verwender und den unterschiedlichen Systemfunktionen widerspiegeln [WHH07]. Diese Arten von Systemen finden sich in der Literatur häufig auch unter dem Namen Workflow-Managementsysteme (WfMS).

Application-to-Application (A2A) Systeme werden zur Integration von Altsystemen eingesetzt. Die Integration umfasst dabei die funktionale Verbindung, also den Aufruf von Funktionen verschiedener Altsysteme in der richtigen Reihenfolge, als auch die Integration verschiedener Datenbestände bzw. die Transformation der Ausgabedaten einer Systemfunktion in das Eingabeformat einer anderen Systemfunktion. Die durch A2A-Systeme unterstützen Prozesse sind also vergleichsweise stark automatisiert.

Unter *Business Process Management Systemen (BPMS)* wird allgemein eine Mischform von P2A und A2A-Systemen verstanden. Diese Systeme sind häufig Verbunde aus mehreren Werkzeugen zur Modellierung von Geschäftsprozes-

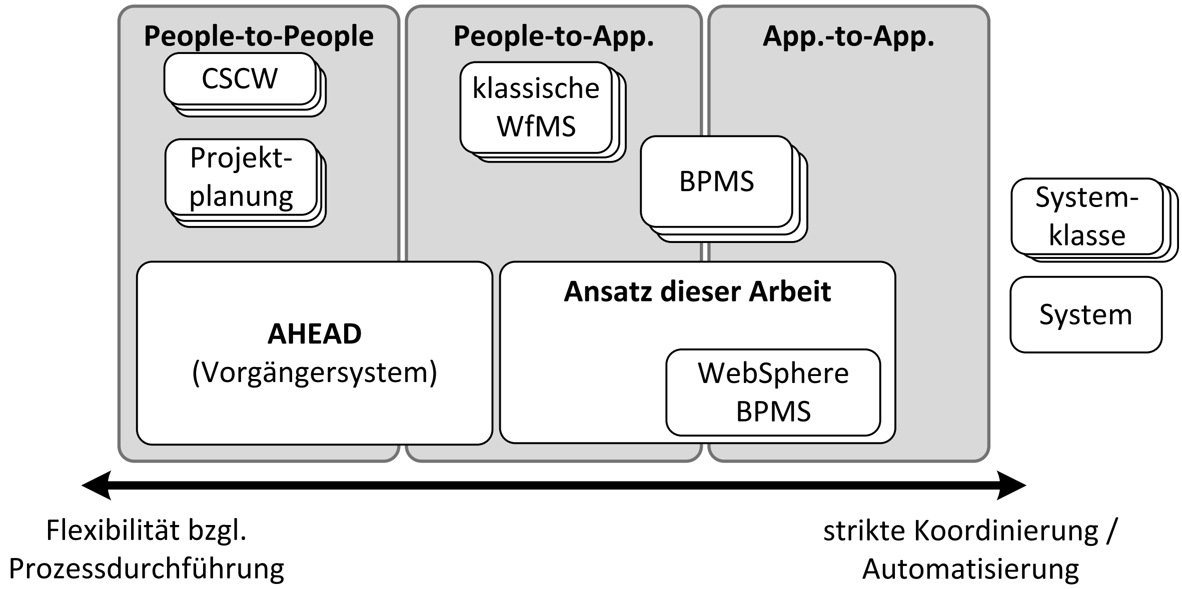


Abbildung 3.2: Einordnung verschiedener Systeme und Systemklassen in die Kategorie P2P bis A2A

sen auf unterschiedlichem Präzisionsniveau sowie zur deren Koordinierung und Überwachung.

In der Abbildung 3.2 finden sich die oben bereits angesprochene Zuordnung von Systemen und Systemklassen zu den Kategorien P2P, P2A und A2A. Die Grenzen zwischen den Kategorien sind nicht strickt. Daher erstrecken sich manche Systeme über zwei Bereiche. Das trifft beispielsweise auf das Vorgängerprojekt AHEAD zu, das zwar auf das Management zwangsweise manuell durchgeführter Entwicklungsprozesse ausgelegt ist, jedoch insoweit mit Entwicklungswerkzeugen integriert werden kann, dass diese automatisch aufgerufen werden, sobald eine zugehörige Aktivität durchgeführt wird.

3.1.3 Eignung für dynamische Prozesse

In Unterabschnitt 2.4.3 wurde die Eignung von Formalismen für Prozessmodellierungssprachen hinblicklich der Eignung zur Unterstützung dynamischer Prozesse diskutiert. Der Formalismus ist allerdings nicht allein ausschlaggebend.

Dynamische Prozesse können einerseits nur durch Prozessmanagementsysteme angemessen unterstützt werden, in denen sich unvorhergesehene Ereignisse während der Prozessdurchführung geeignet abbilden lassen. Andererseits kann die Prozessdurchführung durch die strikte Koordinierung des Prozessmanagementsystems insoweit profitieren, als dass Prozessbeteiligte sich auf ihre eigentlichen Aufgaben konzentrieren können, beispielsweise die Sichtprüfung in einem Versicherungsprozess, ohne sich um die Koordination vor- oder nachgelagerter Aktivitäten kümmern zu müssen. Sind Prozesse auch nur teilweise automatisiert, d.h., werden einzelne Aktivitäten vollautomatisch durchgeführt, so ist eine strikte Koordinierung zwischen den Aktivitäten unabdingbar.

Koordinierungsunterstützung und Flexibilität stehen zueinander im Widerspruch. Ein System, das dem Prozessbeteiligten in jedem Prozesszustand stets eine Menge von sinnvoll ausführbaren Aktivitäten vorschlägt, ist flexibel. Dieser Ansatz wird beispielsweise bei deklarativen Systemen wie DECLARE [PSA07] verfolgt. CSCW-Systeme können in der Hinsicht als Extremform betrachtet werden, da sie mangels Prozessorientierung überhaupt keine Vorgaben bzgl. der Prozessdurchführung machen. Umgekehrt sind derartige Systeme für die Unterstützung automatisierter Prozessteile nicht zu gebrauchen, da hier strikte Vorgaben bzgl. der Aufrufreihenfolgen unabdingbar sind.

3.1.4 IBM WebSphere BPMS

Die ab Abschnitt 3.2 beschriebenen Ergebnisse sind auf Basis des Prozessmanagementsystems IBM WebSphere BMPS erprobt worden. Da auf Details dieses Systems später des Öfteren Bezug genommen wird, werden im Folgenden die wesentlichen Komponenten WebSphere Integration Developer und WebSphere Process Server beschrieben.

Entwicklungszeit: WebSphere Integration Developer

Mit dem WebSphere Integration Developer können insbesondere WS-BPEL-Prozessdefinitionsmodelle in einer graphischen Notation ähnlich zu der von SimBPEL (vgl. Abschnitt 2.3) modelliert werden. Auch für Schnittstellendefinitionen in WSDL stehen dedizierte Editierwerkzeuge bereit.

WS-BPEL-Prozessdefinitionsmodelle verhalten sich technisch wie Implementierungen von WebServices, deren Schnittstellen in WSDL-Dokumenten definiert sind. Die Angaben dieser Dokumente müssen für ein lauffähiges System noch um Bindungsinformationen ergänzt werden, die in Dokumenten gemäß der *Standard Component Architecture (SCA)* [BBBE07] vorgehalten sind. Aus diesen Bindungsinformationen geht insbesondere hervor, unter welcher global eindeutigen Adresse (URL) die Implementierung eines WebServices aufzufinden ist und welche Kommunikationsprotokolle unterstützt werden.

Laufzeit: WebSphere Process Server

Der WebSphere Process Server ist die Laufzeitumgebung für Prozessinstanzen von WS-BPEL-Prozessdefinitionsmodellen. Die einzelnen anhand des

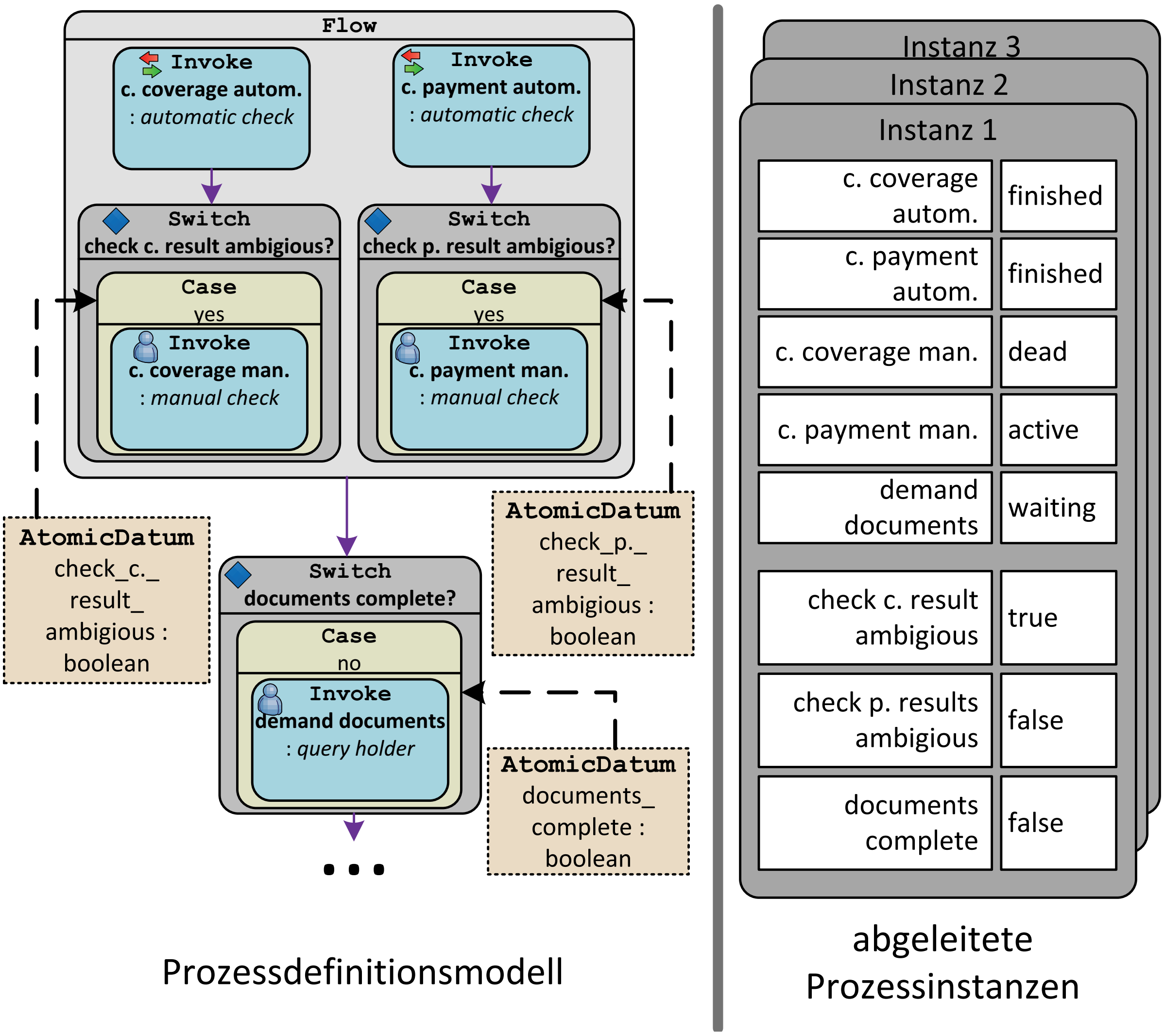


Abbildung 3.3: Technische Trennung zwischen Prozessdefinitionsmodellen und Prozessinstanzen

WebSphere Integration Developers erzeugten Prozessdefinitionsmodelle können in den WebSphere Process Server übertragen werden. Jedes WS-BPEL-Prozessdefinitionsmodell kann dann beliebig häufig instanziiert werden. Die Instanziierung geschieht dabei durch den Aufruf einer von der WS-BPEL-Definition implementierten Exportschnittstellenfunktion.

Die Prozessinstanzen sind komplexe Datenverbunde. In diesen Datenverbunden werden im Wesentlichen nur *Prozesszustände* gespeichert. Zu einem bestimmten Zeitpunkt beinhaltet eine Prozessinstanz die aktuellen Fortschrittsinformationen, also welche Aktivität sich in welchem Zustand befindet, und die aktuelle Belegung der Prozessvariablen. In einer Prozessinstanz ist insbesondere *nicht* die Ablaufstruktur redundant zum Prozessdefinitionsmodell vorhanden. Daher sind Prozessinstanzen im Sinne des WebSphere Process Servers von den Prozessinstanz*modellen*, wie in Kapitel 2 beschrieben, zu unterscheiden. Vielmehr verweisen Prozessinstanzen lediglich auf Prozessdefinitionsmodelle, von denen Sie abgleitet sind. Ablaufstruktur und Zustand eines Prozesses sind im WebSphere Process Server also voneinander technisch getrennt.

Abbildung 3.3 zeigt stark vereinfacht ein Beispiel für die im WebSphere Process Server zur Prozessdurchführung benötigten Daten. Links in der Abbildung ist ein beispielhaftes Prozessdefinitionsmodell dargestellt, rechts befinden sich davon abgeleitet Prozessinstanzen vereinfacht in der Form, wie sie im WebSphere Process Server vorliegen. Aus der Instanz 1 zusammen mit dem Prozessdefinitionsmodell geht beispielsweise hervor, dass in einem bestimmten Prozess die automatischen Deckungsprüfungen (check coverage autom. und check payment autom.) bereits abgeschlossen wurden (finished), die manuelle formelle Deckungsprüfung check payment manually derzeit durchgeführt wird (active), die materielle Deckungsprüfung check coverage manually nicht notwendig ist (dead) und die Nachforderungsaktivität demand documents für zusätzliche Dokumente noch wartend (waiting) ist. Des Weiteren sind in dem dargestellten Fragment drei Prozessvariablen booleschen Typs enthalten, die in der Prozessinstanz wie dargestellt belegt sind.

Diese Entwurfsentscheidung, Definitionen von Instanzen technisch zu trennen, ist in vielen informatischen Systemen zu finden. Eine Instanziierungbeziehung findet sich beispielsweise zwischen Klasse und Objekt in objektorientierten Programmiersprachen. Die Klasse entspricht hierbei dem Prozessdefinitionsmodell, ein Laufzeitobjekt einer Prozessinstanz. Zur Entwicklungszeit objektorientierter Programme wird die Illusion erzeugt, als sei die Klassenimplementierung, also die Definition einzelner Klassenmethoden, in den Objekten redundant zur Klasse vorhanden. Es wird dabei davon abstrahiert, dass die Objekte zur Laufzeit tatsächlich im Wesentlichen nur Werte der in der Klasse deklarierten Instanzvariablen speichern.

Bei Prozessmanagementsystemen führt insbesondere die technische Trennung jedoch zu den bereits beschriebenen Problemen in Dynamiksituationen. Durch die Trennung ist nicht offensichtlich, wie eine dynamische Prozessänderung technisch umzusetzen ist. Prozessinstanzmodelle im Sinne von Abschnitt 2.2 liegen nicht vor. Prozessinstanzen weisen selbst keinerlei Ablaufstruktur auf. Eine Änderung in den Prozessinstanzen selbst ändert nichts an den Ablaufvorgaben des Prozessdefinitionsmodells, eine Änderung im Prozessdefinitionsmodell wirkt sich auf alle zugeordneten Instanzen aus und wird vom WebSphere Process Server verhindert, d.h. von den betreffenden Benutzungs- und Programmierschnittstellen nicht angeboten. Im folgenden Abschnitt wird ein Ansatz beschrieben, mittels dessen dynamische Prozessänderungen trotz der technischen Trennung zwischen Prozessdefinitionsmodell und -instanzen auf Basis des WebSphere Process Server realisiert werden können.

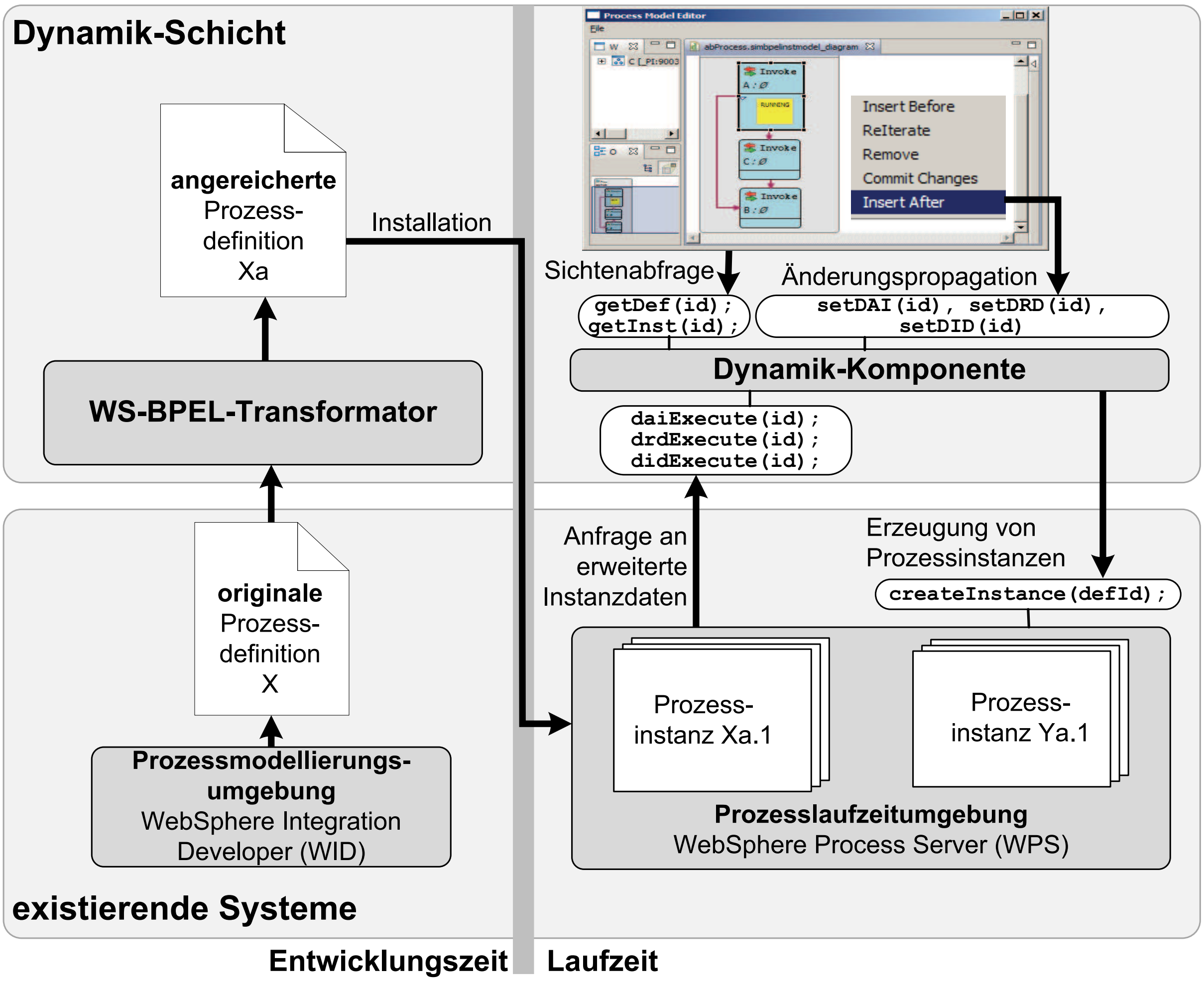


Abbildung 3.4: Grobarchitektur für die Simulation dynamischer Änderungen [WEH08]

3.2 Simulationsansatz

Der WebSphere Process Server unterstützt die in Unterabschnitt 1.3.2 beispielhaft erläuterten Dynamiksituationen in Prozessen nicht. Beispielsweise können in Prozessdefinitionsmodellen nicht modellierte Aktivitäten nachträglich, d.h. zur Prozesslaufzeit nicht eingefügt werden. Der Hauptgrund hierfür liegt in der typischen strikten Trennung zwischen Prozessdefinitionsmodell und -instanz innerhalb des WebSphere Process Servers (vgl. Unterabschnitt 3.1.4).

Eine Zielsetzung dieser Arbeit war, diese Beschränkung zu überwinden, ohne den WebSphere Process Server als technische Grundlage zu ersetzen. Daher wurde ein Simulationsansatz gewählt, der die Implementierung der existierenden Systeme (gezwungenermaßen) unverändert lässt, aus Sicht von Prozessbeteiligten aber die Illusion erzeugt, die Vorgaben in Prozessdefinitionsmodellen für einen Prozessbeteiligten seien zu Prozesslaufzeit noch nachträglich änderbar.

Abbildung 3.4 zeigt die Grobarchitektur, die dem Simulationsansatz zugrunde liegt. Die bestehenden Systeme WebSphere Integration Developer und WebSphere Process Server bleiben hierbei unverändert. Prozessdefinitionsmodelle in WS-BPEL, die in der Entwicklungsumgebung WebSphere Integration Developer modelliert werden, werden mittels eines WS-BPEL-Transformators vollautomatisch um dynamiksimulierende Strukturen erweitert. Diese zusätzlichen Strukturen sind Kombinationen von WS-BPEL-Elementen, die in Unterabschnitt 2.1.5 beschrieben wurden. Diese sorgen auf der Seite der Prozessdefinitionsmodelle dafür, dass die im nächsten Abschnitt beschriebenen Änderungsoperationen möglich werden. Die Erweiterung auf Entwicklungszeitseite wird durch eine Erweiterung auf Laufzeitseite ergänzt. Die *Dynamik-Komponente* erweitert die in den Prozessinstanzen vorhandenen Zustandsinformationen. Über diese zusätzlichen Informationen kann der Kontrollfluss innerhalb der erweiterten Prozessinstanz so gesteuert werden, dass für Prozessbeteiligte der Eindruck entsteht, Änderungen an der Ablaufstruktur seien zur Prozesslaufzeit möglich. Das heißt, dass Prozessbeteiligte strukturelle, dynamische Änderungen in Prozessinstanzmodellen vornehmen können, die auf Änderungen der erweiterten Instanzdaten im WebSphere Process Server abgebildet werden.

3.3 Muster für Dynamiksituationen

Die Unterstützung für dynamische Prozessänderungen folgt in dieser Arbeit Mustern, sogenannten *Änderungsmustern*. Diese Muster beziehen sich auf bestimmte Arten von dynamischen Änderungen in Prozessinstanzmodellen, d.h. ein Muster steht für eine bestimmte Art dynamischer Prozessänderung.

Die hier behandelten Muster sind das dynamische Einfügen von Prozessteilen in einen laufenden Prozess, das dynamische Entfernen sowie das Wiederholen bestimmter Prozessteile durch dynamische Rücksprünge. Im Vergleich zu den Änderungsmöglichkeiten, die ein Prozessmodellierer bei der Bearbeitung von Prozessdefinitionsmodellen hat, sind die Änderungsmuster für dynamische Prozessänderungen beschränkter. Insbesondere ist über die Änderungsmuster nicht möglich, neue alternative Ausführungszweige zu modellieren. Stattdessen können nur bestehende Ausführungszweige manipuliert werden. Dies trifft jedoch die typischen Anforderungen von Prozessbeteiligten, die durch dynamische Änderungen nur auf eine bestimmte, prozessspezifische Dynamiksituation reagieren, die die Änderung eines bestehenden Ablaufs erforderlich machen. Die Modellierung, auf welche Arten ein bestimmter Prozess in anderen Fällen auch durchgeführt werden könnte, ist Aufgabe des Prozessmodellierers, nicht eines Prozessbeteiligten.

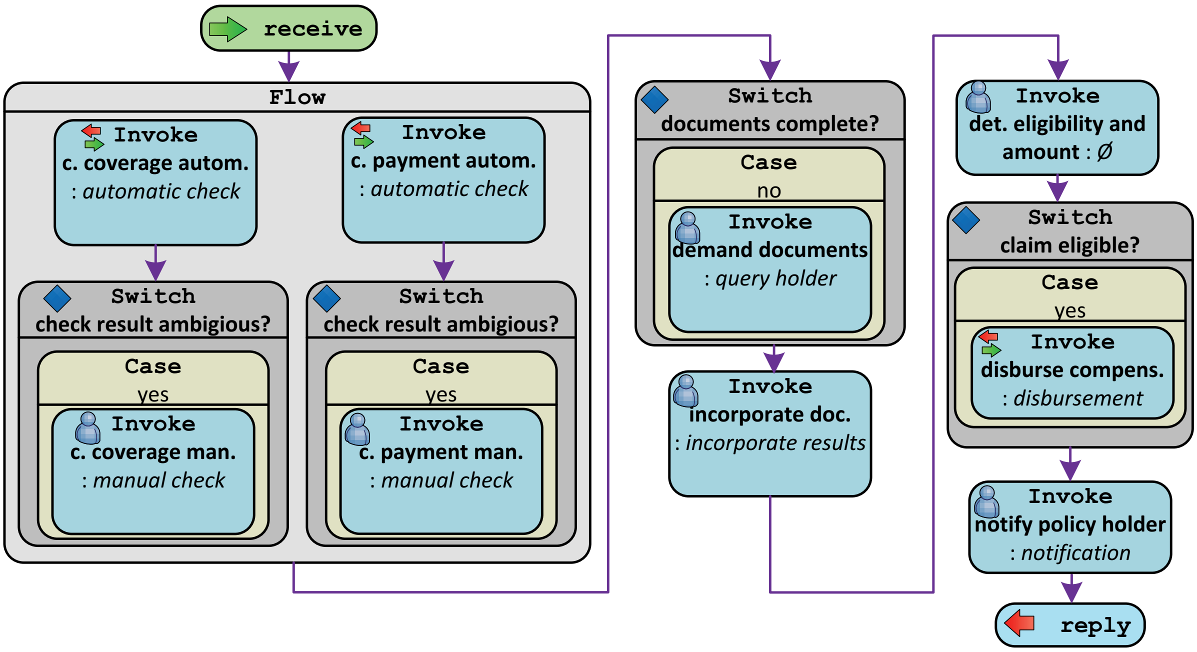


Abbildung 3.5: Ursprüngliches Prozessdefinitionsmodell

Die Änderungsmuster *korrespondieren* direkt mit den Operationen, die einem Prozessbeteiligten als dynamische Änderungsoperationen für Prozessinstanzmodelle eines laufenden Prozesses zur Verfügung gestellt werden. Bezogen auf die Syntax der Prozessinstanzmodelle sind die Änderungsoperationen *komplex*. Beispielsweise beim dynamischen Einfügen oder Löschen muss sich ein Prozessbeteiligter nicht um Detailänderungen an der durch Links gebildeten Struktur kümmern, die im unmittelbaren Kontext der eingebetteten bzw. entnommenen Aktivität zwangsweise stattfindet. Das dynamische Einfügen einer Aktivität C zwischen zwei durch Links verbundenen Aktivitäten A und B findet aus Sicht des Prozessbeteiligten in einem Schritt statt, statt in vier Schritten, die aus (1) dem Einfügen von C in den Prozess besteht, (2) der Verbindung von A und C mit einem Link, (3) der Verbindung von C und B mit einem Link und der (4) Löschung des ursprünglichen Links zwischen A und B.

Muster spielen im Bereich von Prozessmanagementsystemen ganz allgemein eine gewichtige Rolle. Es gibt eine Reihe von Forschungsarbeiten, die sich mit Mustern innerhalb statischer Prozessdefinitionsmodelle beschäftigen [ABHK00, AtHKB03, RHEA05, RAHE05]. Diese Muster dienen in erster Linie als Vergleichskriterien bei der Untersuchung der Ausdruckmächtigkeit verschiedener Sprachen für Prozessdefinitionsmodelle. Im Bereich dynamischer Prozessmanagementsysteme gibt es Arbeiten [WRR07a, WRR07b], die Muster dynamischer Änderungen an Prozessinstanzmodellen studieren.

Die Bedeutung von Mustern ist in dieser Arbeit nochmals stärker, weil mögliche dynamische Änderungen nicht nur anhand dieser Muster klassifiziert werden. Vielmehr baut das hier verfolgte Realisierungskonzept für die Unterstützung dynamischer Änderungen auf Basis des statischen Prozessmanagementsystems WebSphere Process Server von Grund auf Änderungsmustern auf.

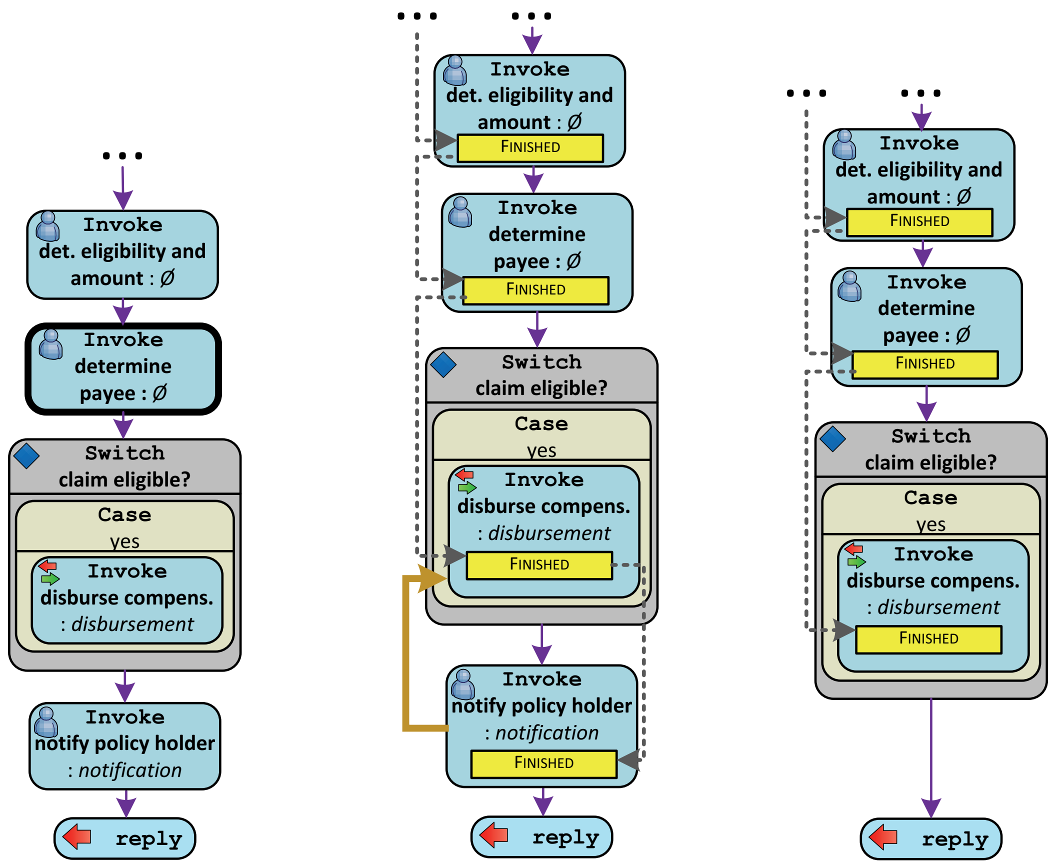
In den folgenden Unterabschnitten werden die unterstützten Änderungsmuster an Beispielen erläutert. Ausgangspunkt ist dabei das Prozessdefinitionsmodell aus Abbildung 3.5. Dieses ist aus Gründen der Übersichtlichkeit gegenüber dem Prozessdefinitionsmodell aus Abbildung 1.8 etwas vereinfacht. Dieses Prozessdefinitionsmodell wird instanziiert. Die Abbildungen 3.6(a) bis 3.6(c) in den nachfolgenden Unterabschnitten stellen Prozessinstanzmodelle dar, die nicht nur den Fortschritt des Prozesses widerspiegeln sondern insbesondere strukturell gegenüber dem ursprünglichen Prozessdefinitionsmodell geändert wurden.

In Unterabschnitt 1.3.2 wurden drei verschiedene Dynamiksituationen und entsprechende dynamische Änderungen in Prozessinstanzmodellen erläutert. Diese sind das dynamische Einfügen und Löschen von Aktivitäten sowie dynamische Rücksprünge. Abbildungen 3.6(a) bis 3.6(c) greifen diese Beispiele in leicht abgewandelter Form auf: Die Änderungen finden in der Exkasso-Phase des Prozesses statt. Dadurch können die folgenden Detailerläuterungen auf ein kleines Prozessfragment beschränkt bleiben. Im Unterschied zu den Beispielen in Unterabschnitt 1.3.2 wird determine payee der Einfachheit halber vor der `Switch`-Aktivität claim eligible eingefügt und der Rücksprung von notify holder zu disburse compensation aus durchgeführt.

3.4 Realisierungen der Dynamikmuster

Abbildung 3.7 stellt Ausschnitte konkreter Prozessdefinitionsmodelle und -instanzen in den Vordergrund, die in Abbildung 3.4 ausgeblendet wurden. Die Prozessinstanzmodelle rechts unten vereinfachen der besseren Darstellung halber die Situation: Wie in Unterabschnitt 3.1.4 erläutert sind Prozessinstanzmodelle in dieser Form gar nicht im WebSphere Process Server vorhanden, sondern ergeben sich indirekt aus Prozessinstanzen (Datenverbunde) zzgl. dem jeweils zugehörigen Prozessdefinitionsmodell. Links ist die Wirkungsweise des WS-BPEL-Transformators zu sehen. Dieser erweitert ein originales Prozessdefinitionsmodell X um weitere Aktivitäten und liefert als Ergebnis ein angereichertes Prozessdefinitionsmodell Xa.

Die vom WS-BPEL-Transformator erzeugten zusätzlichen Aktivitäten werden im Folgenden *dynamikerzeugende Muster* genannt. Es handelt sich um `Flow`-, `Invoke`-, `Switch`- und `While`-Aktivitäten, über die der Kontrollfluss zur Laufzeit so gesteuert werden kann, dass dynamische Änderungsoperationen simuliert



(a) Dynamische Einfügung (b) Dynamischer Rücksprung (c) Dynamische Löschung

Abbildung 3.6: Beispielhafte dynamische Änderungen

werden können. Zur Laufzeit werden dabei erweiterte Instanzdaten bei Ausführung dieser Aktivitäten ausgewertet, die in der Dynamik-Komponente vorgehalten werden.

Beispiel 3.1 (Dynamisches Einfügen) Die folgenden Erläuterungen beziehen sich auf Abbildung 3.7 und skizzieren die Idee des Ansatzes im Groben am Beispiel einer dynamischen Einfügung wie in Abbildung 3.6(a).

1. Der WS-BPEL-Transformator reichert ein vormodelliertes Prozessdefinitionsmodell um DAI-Flow- und DAI-Invoke-Aktivitäten an. Während der Durchführung der Prozessinstanz XA.1 werde nun eine Aktivität determine payee dynamisch eingefügt. Dies geschieht über den Prozessmodelleditor, der in Abbildung 3.7 ausgelassen wurde.

2. Die dynamische Einfügung bewirkt in der Dynamik-Komponente, dass die Bindungsinformationen für die Aktivität DAI1 innerhalb der Instanz

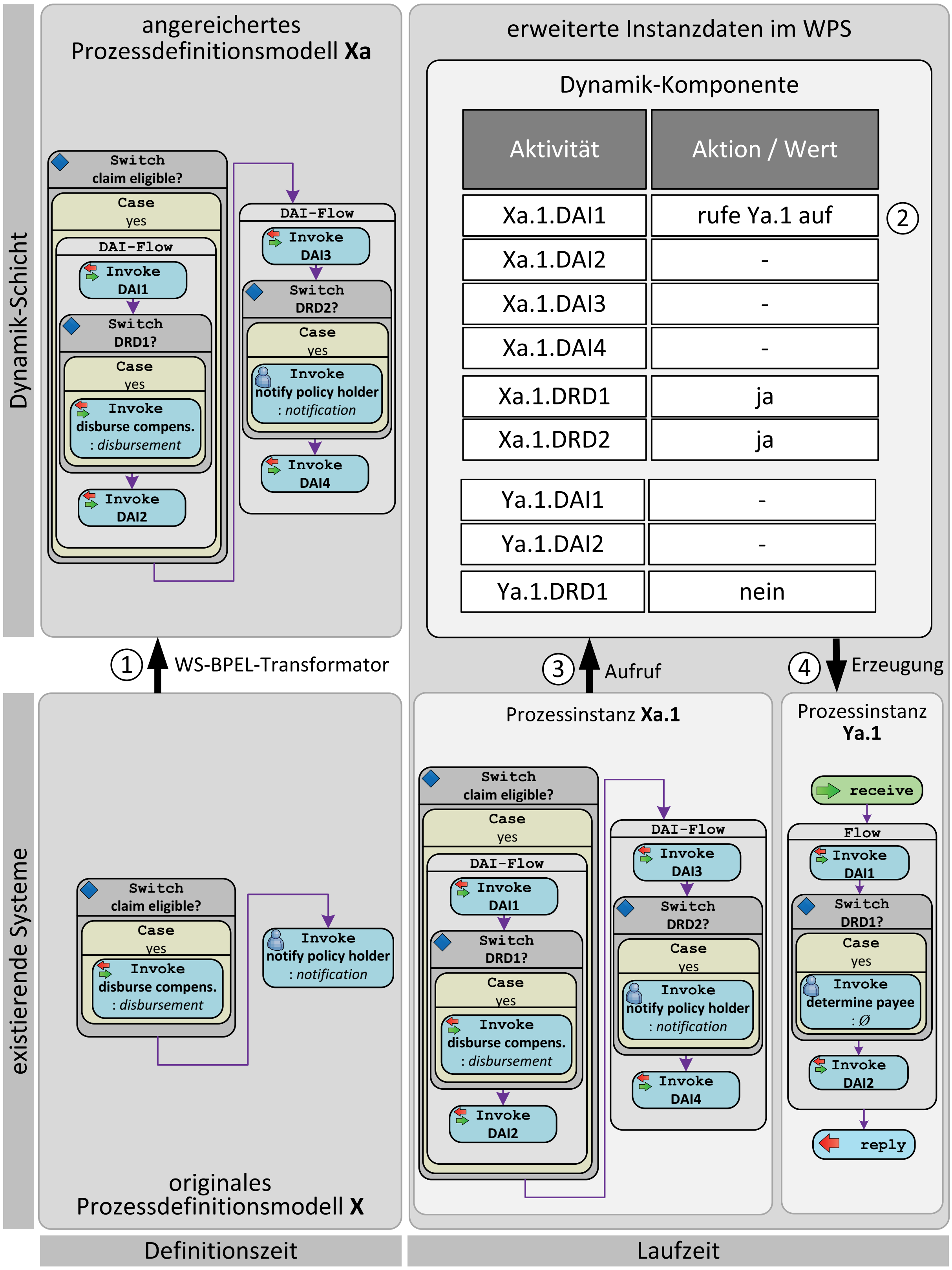


Abbildung 3.7: Prinzip und Wirkung dynamikerzeugender Muster

Xa.1 so geändert wird, das diese Aktivität eine Prozessinstanz Ya.1 erzeugt und aufruft.

3. Erreicht der Kontrollfluss in Xa.1 nun DAI1, so ruft diese eine Standardfunktion der Schnittstelle der Dynamik-Komponente auf.

4. Innerhalb der Dynamik-Komponente wird dann der Aufruf gemäß des Eintrags in der Tabelle an eine neu erzeugte Prozessinstanz Ya.1 delegiert. Nach Durchführung von Ya.1 wird DAI1 in Xa.1 finalisiert und der Kontrollfluss fährt in Xa.1 fort.

Offensichtlich funktioniert dynamisches Einfügen insoweit rekursiv, als auch in Ya.1 wiederum DAI-Invoke-Aktivitäten vorhanden sind und als Aussprungspunkt in weitere Prozessinstanzen dienen können.

Die folgenden Unterabschnitte erläutern die Generierung und Wirkung dynamikerzeugender Muster im Detail. Die Erläuterungen beziehen sich nicht auf die letztendliche Implementierung der Dynamik-Schicht als Erweiterungen des WebSphere Process Server sondern auf Vorüberlegungen, die unabhängig von der technischen Basis des WebSphere Process Servers anhand von Graphgrammatiken durchgeführt wurden. Die Ausrichtung der Erklärungen an Graphgrammatiken hat den Vorteil, in ihnen von rein technischen Detailentscheidungen und aus dem WebSphere Process Server stammenden technischen Zwängen absehen zu können, die das Verständnis der grundlegenden Konzepte erschweren. Zudem beinhalten die Graphzustände stets alle relevanten Ausführungsinformationen, die in der tatsächlichen Implementierung auf verschiedene Artefakte und Speicherbereiche aufgeteilt sind (vgl. Diskussion in Unterabschnitt 2.4.3).

Zwei Graphersetzungsregelmengen spielen in den folgenden Erläuterungen eine Rolle:

$\mathcal{R}_{trans}$ ist die Regelmenge, die Prozessdefinitionsmodelle als Prozessgraphen um dynamikerzeugende Muster anreichert. Die Graphersetzungsregeln der Abbildungen 3.9, 3.13 und 3.17 stammen aus dieser Regelmenge.

$\mathcal{R}_{bpel}$ wurde in Unterabschnitt 2.5.3 eingeführt. Über sie wird die Ausführungssemantik von Prozessdefinitions- und -instanzmodellen spezifiziert. Da Zustandsgraphen im von dieser Graphgrammatik erzeugten Graphtransitionssystem Ausführungszustände widerspiegeln, ist sie geeignet, von Prozessbeteiligten durchgeführte Eingriffe in die Ausführungszustände darzustellen.

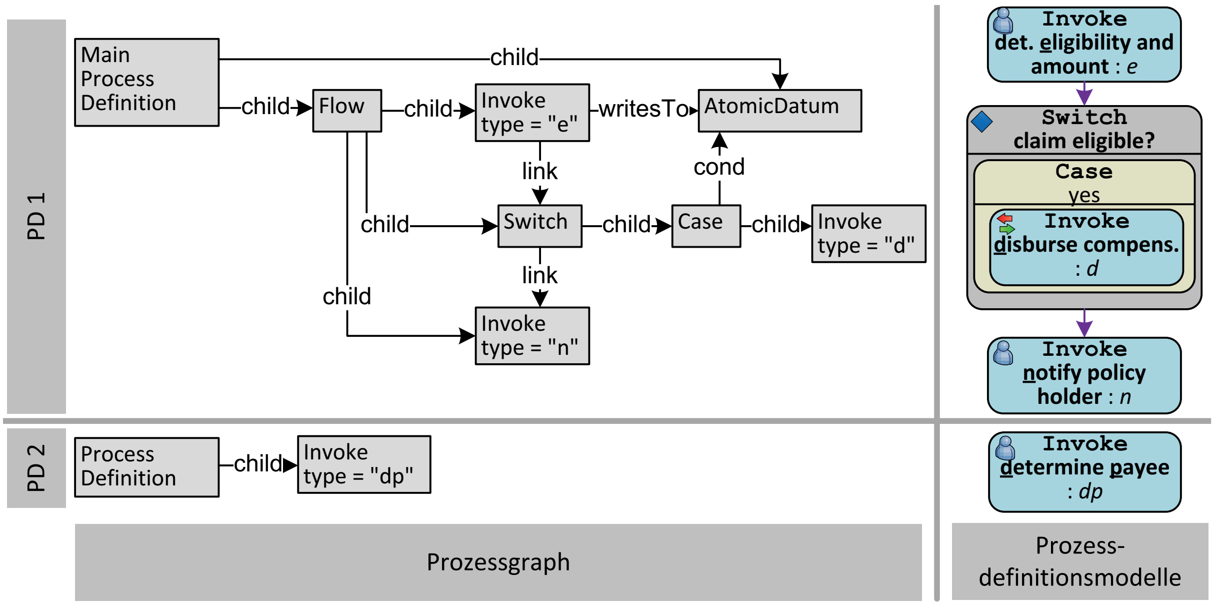


Abbildung 3.8: Ausgangszustand mit zwei Prozessdefinitionsmodellen

In Abbildung 3.8 sind rechts die zwei Prozessdefinitionsmodelle dargestellt, die auch im Beispiel von Abbildung 3.5 enthalten sind. Der entsprechende, links abgebildete Prozessgraph dient als durchgängiges Beispiel bei den folgenden Detailerklärungen zu dynamikerzeugenden Mustern.

3.4.1 Dynamisches Einfügen

Die folgenden Erläuterungen sind gegliedert in, erstens, die Anreicherung der Aktivitätsstruktur um dynamikerzeugende Muster in Prozessgraphen und, zweitens, die Ausnutzung der dynamikerzeugenden Muster zur Prozesslaufzeit.

Prozessdefinitionstransformation

Idee der Anreicherung ist, für jede Aktivität in einem ursprünglichen Prozessdefinitionsmodell X zwei weitere Invoke-Aktivitäten im angereicherten Prozessdefinitionsmodell Xa einzuführen, die zur Prozesslaufzeit als Aus- und Wiedereinsprungspunkte dienen. Diese speziellen Invoke-Aktivitäten werden im Folgenden mit *Dynamic Adding Invocations (DAI)* bezeichnet. Jede Aktivität des ursprünglichen Prozessdefinitionsmodells wird ersetzt durch eine Flow-Aktivität. Diese DAI-Flow-Aktivität beinhaltet die ursprüngliche Aktivität und zwei mittels Links seriell vor- und nachgelagerte DAI-Invoke-Aktivitäten.

Diese Erweiterung ist formal als Graphersetzungsregel addDAI spezifiziert, die auf Prozessgraphen angewendet wird wie den links in Abbildung 3.8. Zentral in der Regel ist der Knoten @. Dieser passt auf Aktivitäten des ursprüngli-

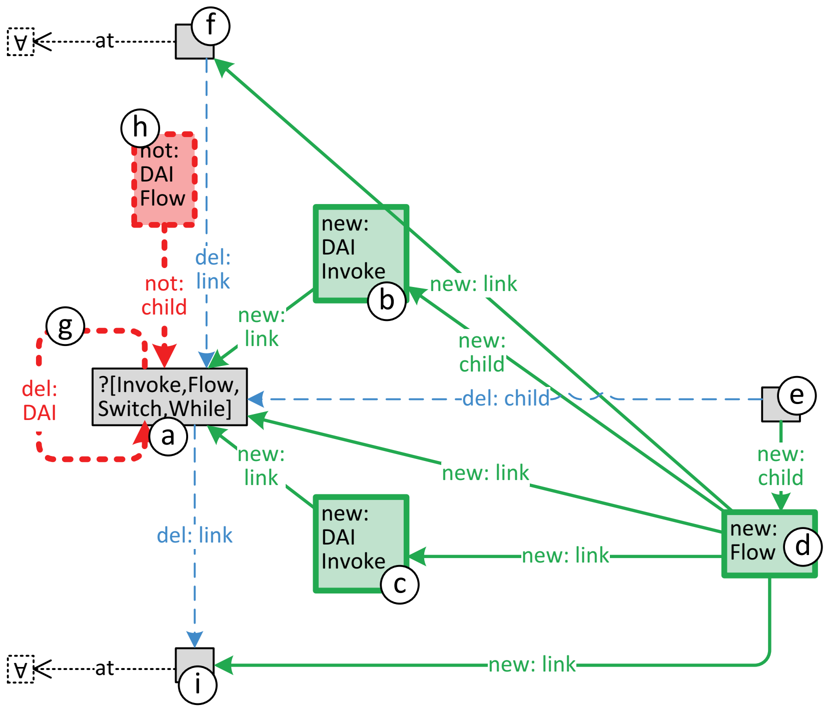


Abbildung 3.9: Graphersetzungsregel addDAI zur Anreicherung um “Dynamic Adding Invocations”

chen Prozessdefinitionsmodells. Diese haben keine Knotenbeschriftung DAI, daher die negative Anwendungsregel ⓖ. Außerdem darf der Knoten nicht schon behandelt worden sein, was der Fall ist, wenn er Kind eines mit DAI und Flow beschrifteten Knoten ⓗ ist. Die Anwendung von addDAI bewirkt nun, dass zwei neue Knoten ⓑ und ⓒ erzeugt werden und diese link-Vorgänger (ⓑ) bzw. -Nachfolger (ⓒ) von ⓐ werden. Gleichzeitig wird ein DAI-Flow-Knoten erzeugt, der der neue Vater bzgl. child-Beziehung von ⓐ ist. Der ursprüngliche Vater ⓔ wird Großvater. Darüber hinaus werden vormalige link-Vorgänger ⓕ und -Nachfolger ⓘ von ⓐ nun zu Vorgängern des neuen Vaters ⓓ.

Beispiel 3.2 (Anwendung von addDAI) Wird die Graphersetzungsregel addDAI auf den Prozessgraphen aus Abbildung 3.8 angewendet und zwar im Speziellen so, dass der Musterknoten ⓐ auf den e-Knoten passt, ergibt sich der Graphzustand aus Abbildung 3.10. Man sieht, dass durch die Regelanwendung nun der mit DAI und Flow beschriftete Knoten die Stelle des ursprünglichen Knotens einnimmt. Selbstverständlich wird addDAI generell auf sämtliche ursprünglichen Aktivitäten angewendet. Der hieraus resultierende Prozessgraph beinhaltet dementsprechend viele Knoten und ist daher aus Gründen der Übersichtlichkeit hier nicht wiedergegeben.

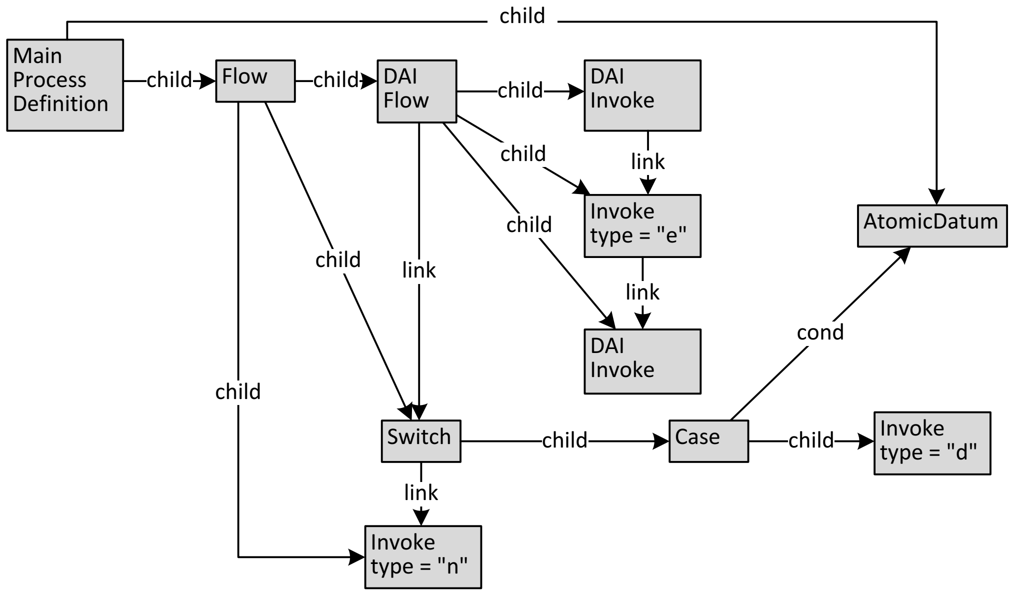


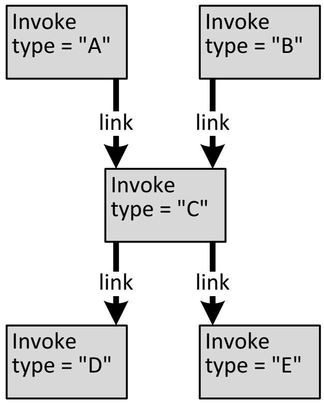
Abbildung 3.10: Prozessgraph nach Anwendung von addDAI auf e-Aktivität

Der Prozessgraph aus Abbildung 3.8 taugt nur bedingt zur Demonstration der Anwendung der addDAI-Regel in komplizierten Linkstrukturen. Abbildung 3.11(a) stellt einen Ausschnitt eines Prozessgraphen mit einer Aktivitätsstruktur dar, die im Wesentlichen aus einem Flow und fünf darin enthaltenen Invoke-Aktivitäten besteht. Abbildung 3.11(b) zeigt den Prozessgraphen nach Anwendung von addDAI auf alle passenden Anwendungsstellen.

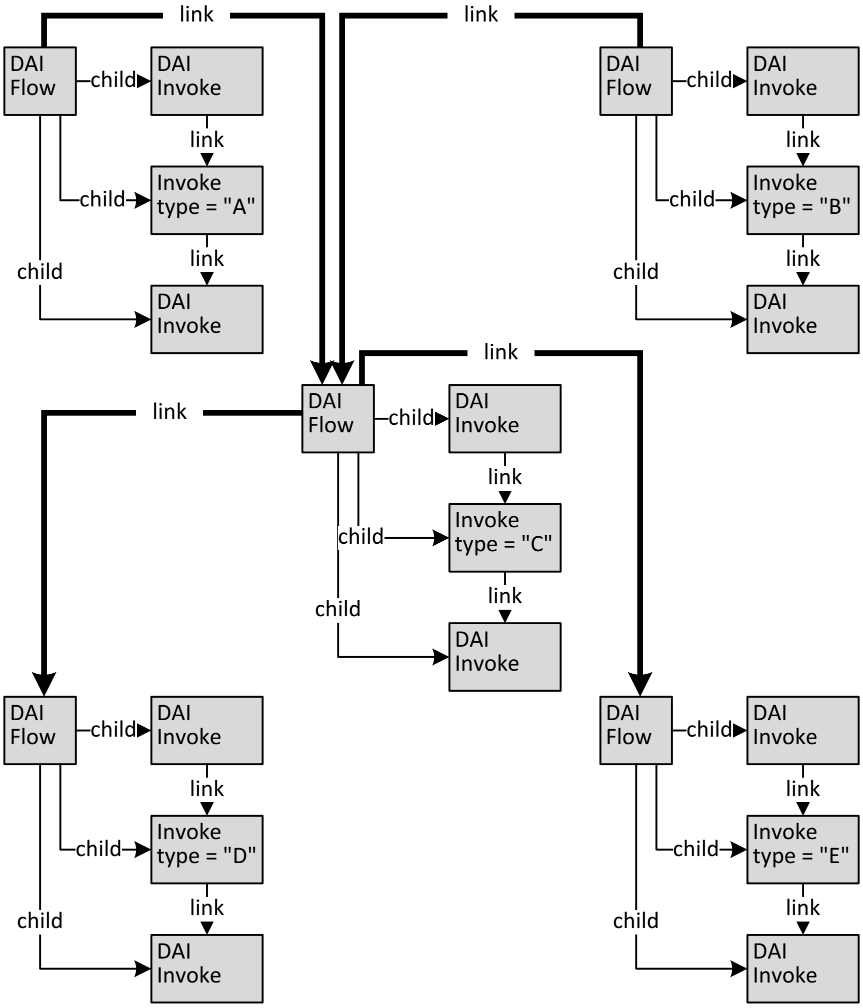
Am Beispiel von Abbildung 3.11 zeigt sich auch, dass im allgemeinen Fall pro Invoke-Aktivität im ursprünglichen Modell jeweils eine vor- und eine nachgelagerte DAI-Aktivität notwendig ist. Beispielsweise das dynamische Einfügen einer X-Aktivität nach der A-Aktivität ist hier gerade nicht gleichbedeutend mit dem Einfügen vor der C-Aktivität. Ein Einfügen vor der C-Aktivität bewirkt die Neubindung der DAI-Aktivität, die C direkt vorgelagert ist. Demzufolge kann die X-Aktivität erst ausgeführt werden, wenn sowohl die A- als auch die B-Aktivität finalisiert wurden. Ein Einfügen nach beispielsweise der A-Aktivität betrifft hingegen die der A-Aktivität nachgelagerten DAI-Aktivität. Entsprechend muss auch nur A finalisiert werden, damit X ausgeführt werden kann, jedoch nicht unbedingt auch B.

Eingriffe zur Prozesslaufzeit

Die Anwendungen der Regeln aus $\mathcal{R}_{bpel}$ (vgl. Unterabschnitt 2.5.3) auf den Prozessgraphen in Abbildung 3.10 führen zu einem Zwischenzustand wie in Abbildung 3.12 dargestellt (jedoch zunächst ohne invokes-Kante). Die DAI-Knoten können zur Prozesslaufzeit genutzt werden, um das dynamische Einfügen zusätzlicher Aktivitäten zu simulieren. Insbesondere kann hierdurch



(a) Ursprünglicher Prozessgraph mit komplizierter Linkstruktur



(b) Prozessgraph nach mehrfacher Anwendung von addDAI

Abbildung 3.11: Anwendung von addDAI auf komplizierte Linkstrukturen

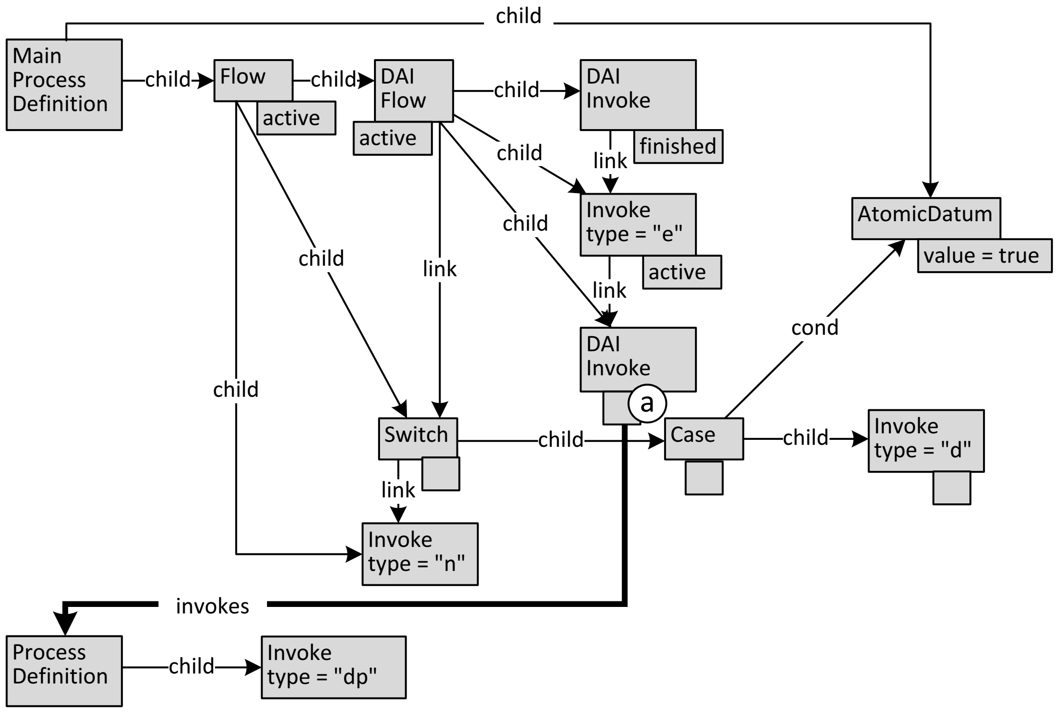


Abbildung 3.12: Neubindung im Prozessgraphen

ein dynamisches Einfügen der dp-Aktivität (determine payee) nach der e-Aktivität (determine eligibility and amount) realisiert werden.

Hierzu wird in den Prozessgraphen insoweit eingegriffen, als dass der Instanzknoten ⓐ per invokes-Kante mit dem Teilprozessgraphen verbunden wird, der das Prozessdefinitionsmodell darstellt, welches die dp-Aktivität beinhaltet.

Der händische Eingriff im Prozessgraphen ist derart, dass er in der eigentlichen Implementierung auf Basis des WebSphere Process Server nachgebildet werden kann. Der Eingriff entspricht gerade einer dynamischen Neubindung wie sie in der Dynamik-Komponente vorgenommen werden kann (vgl. ③ in Beispiel 3.1). Die viel naheliegendere direkte Änderung der Ablaufstruktur ist zwar im Prozessgraphen ohne Weiteres ebenfalls möglich, technisch dann aber nicht mehr auf den auf dem WebSphere Process Server basierenden Ansatz übertragbar.

3.4.2 Dynamisches Löschen

Prozessdefinitionstransformation

Die Idee dieser Erweiterung ist, jede `Invoke`-Aktivität A eines ursprünglichen Prozessdefinitionsmodells X im angereicherten Prozessdefinitionsmodell Xa in eine neue `Switch`-Aktivität einzuschachteln. Diese speziellen `Switch`-

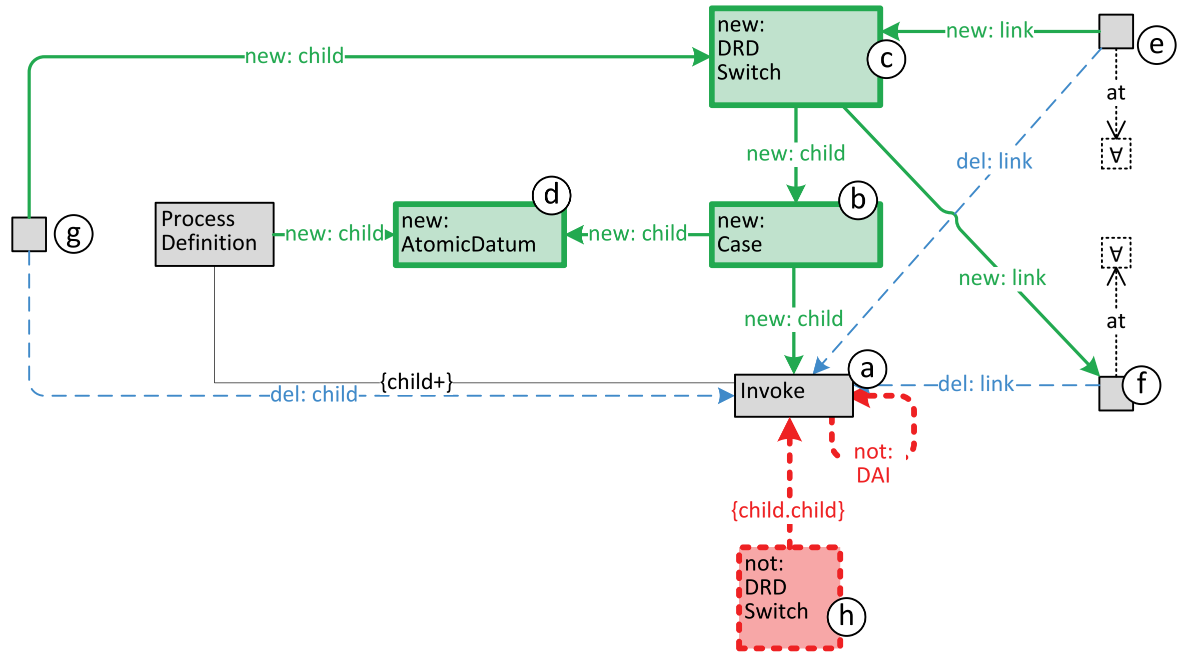


Abbildung 3.13: Graphersetzungsregel addDRD zur Anreicherung um “Dynamic Removing Decisions”

Aktivitäten dienen zur Prozesslaufzeit dazu, den Kontrollfluss ggf. an der Aktivität A vorbeizulenken, indem die Ausführung des einzigen Zweiges (Case) der neuen umgebenen Switch-Aktivität durch entsprechende Belegung der Zweigbedingung verhindert wird. Im Folgenden werden diese eingeführten Switch-Aktivitäten *Dynamic Removing Decisions (DRD)* genannt.

Die Graphersetzungsregel addDRD aus Abbildung 3.13 beschreibt die Anreicherung um DRD-Aktivitäten formal. Die einzuschachtelnde Invoke-Aktivität ist die mit ⓐ markierte. Sie wird eingeschachtelt, d.h. wird im Prozessgraphen Kind (child) eines Zweiges ⓑ, der selbst Kind der neuen Switch-Aktivität ⓒ ist. Ob der Zweig zur Prozesslaufzeit ausgeführt wird oder nicht, wird durch eine neue Prozessvariable ⓓ gesteuert. In der Link-Struktur des ursprünglichen Prozessdefinitionsmodells nimmt die DRD-Aktivität die Stelle der eingeschachtelten Invoke-Aktivität ein. Dies wird durch Löschung der ursprünglich inzidenten Links und Neuerzeugung von Links inzident zur DRD-Aktivität mit gleichen Vorgängern ⓔ und Nachfolgern ⓕ bewerkstelligt. Der ursprüngliche Vater bzgl. child-Kante ⓖ wird Vater der neuen DRD-Aktivität. Die negative Anwendungsbedingung ⓗ verhindert die Mehrfachbehandlung einer Invoke-Aktivität.

Beispiel 3.3 (Anwendung von addDRD) Die einmalige Anwendung der Graphersetzungsregeln addDRD auf den Prozessgraphen aus Abbil-

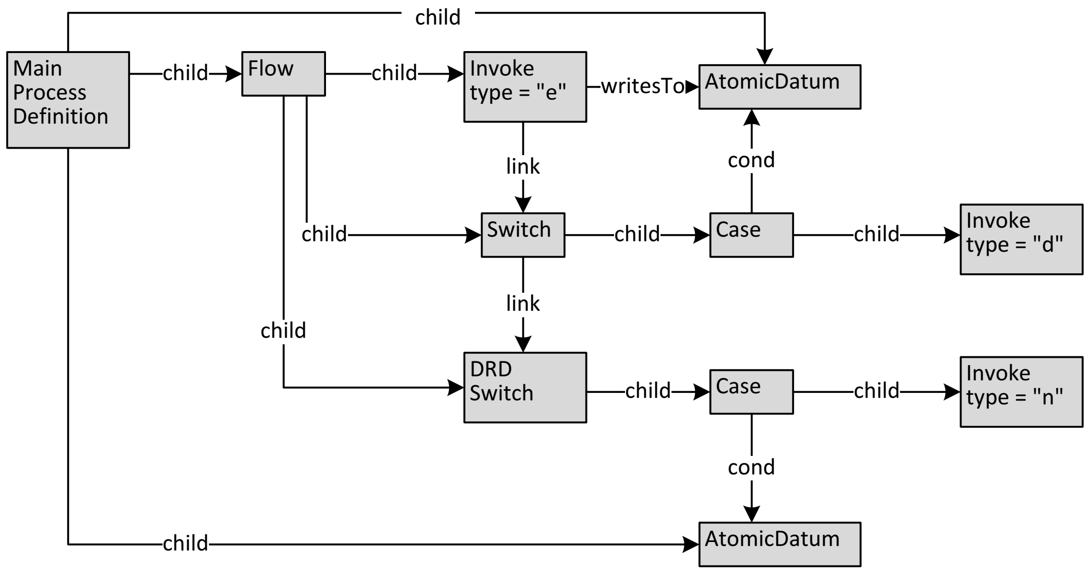


Abbildung 3.14: Prozessgraph nach einmaliger Anwendung von addDRD auf n-Aktivität

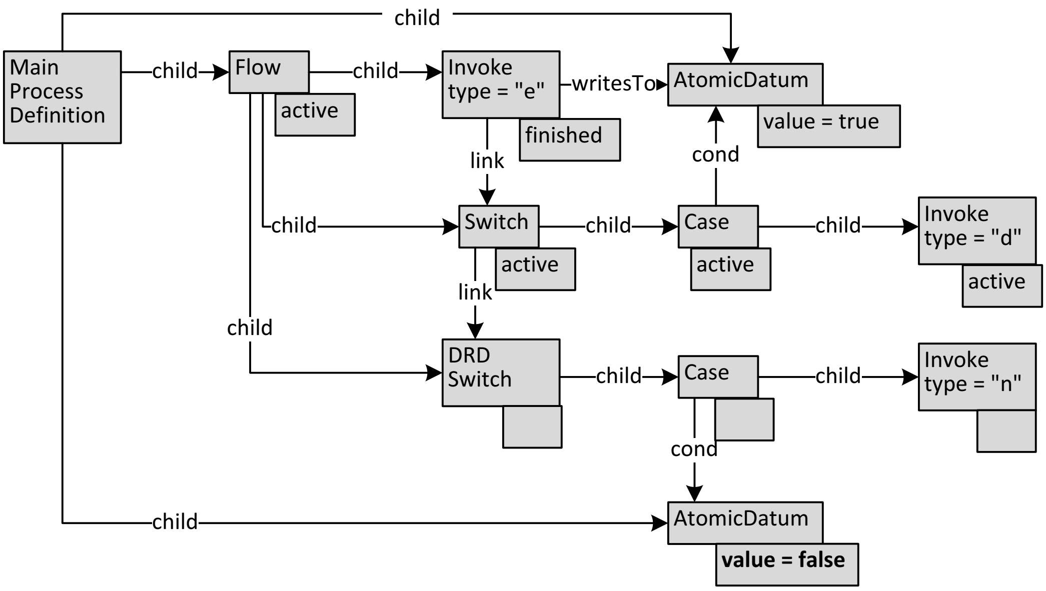
dung 3.8 führt zu einem Prozessgraphen wie in Abbildung 3.14 gezeigt. Die Regel wurde so angewendet, dass der mit @ markierte Knoten auf den Invoke-Knoten n passt. Auch hier ist wieder der Übersichtlichkeit halber nur eine der möglichen Regelanwendungen von addDRD auf den originalen Prozessgraphen gezeigt.

Eingriffe zur Prozesslaufzeit

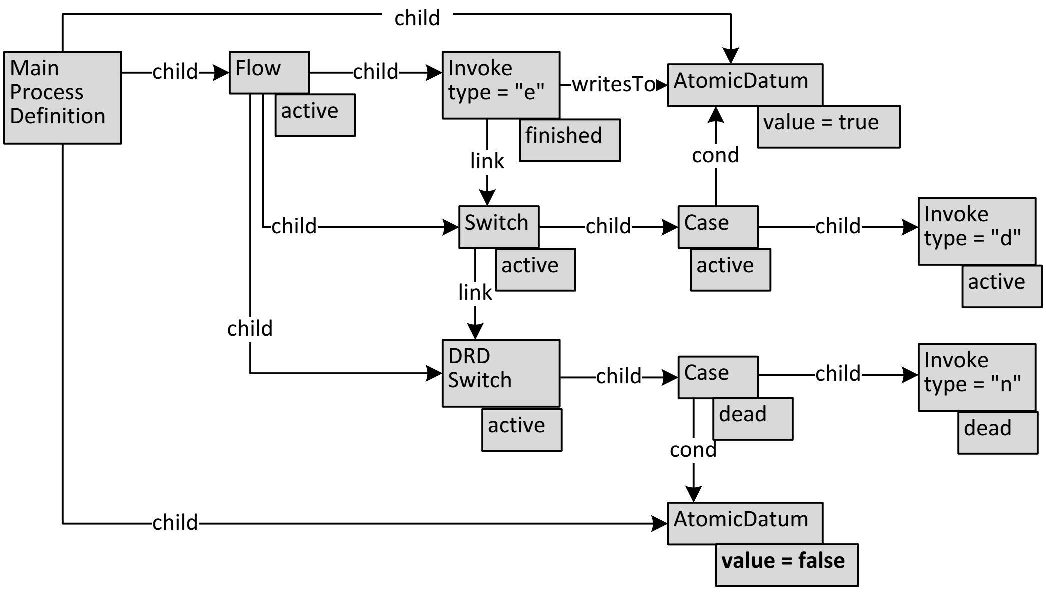
Die Erweiterung des Prozessdefinitionsmodells dienen auch in diesem Fall dazu, dynamische Änderungen zur Prozesslaufzeit durch solche Änderungen zu bewerkstelligen, die sich auch auf Basis des WebSphere Process Server realisieren lassen. Im Fall von DRDs wird der Wert des bei der Transformation erzeugten Prozessdatums geändert.

In Abbildung 3.15 ist dargestellt, wie die hinzugenerierten DRD-Strukturen zur Prozesslaufzeit zur Simulation einer dynamischen Löschung ausgenutzt werden können. Die Abbildung 3.15(a) zeigt den Graphzustand während der Ausführung von Aktivität d. Der Wert des Datums unten in der Unterabbildung wird in diesem Zustand auf false geändert. Als Konsequenz dieser Änderung wird im späteren Prozessverlauf der Zweig, der Aktivität n beinhaltet, nicht ausgeführt, wie Abbildung 3.15(b) beispielhaft zeigt .

Die Transformation und der Eingriff sind wieder so gewählt, dass sie technisch auf eine Implementierung auf Basis des WebSphere Process Servers



(a) Dynamische Änderung während Ausführung von d



(b) Effekt des Eingriffs bei weiterer Prozessausführung

Abbildung 3.15: Bedingungsänderung im Prozessgraphen zur Simulation der dynamischen Löschung der n-Aktivität

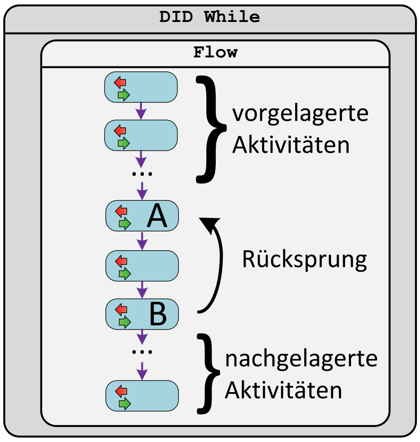


Abbildung 3.16: Realisierungsidee für dynamische Rücksprünge

übertragbar sind. Die naheliegendere direkte Löschung des Knotens bzw. der Aktivität n ist technisch auf Basis des WebSphere Process Server nicht möglich.

3.4.3 Dynamische Rücksprünge

Dynamische Rücksprünge erlauben die Wiederholung bereits aufgeführter Aktivitäten, ohne dass dieses im originalen Prozessdefinitionsmodell, beispielsweise durch die Verwendung von Schleifen, vorgesehen ist. Nachfolgend werden wiederum die notwendige Transformation des Prozessdefinitionsmodells sowie die Eingriffe zur Laufzeit anhand des per Graphersetzungsregel transformierten Prozessgraphs erläutert.

Prozessdefinitionstransformation

Die Idee der Erweiterung zur Unterstützung von dynamischen Rücksprüngen ist, die gesamte Aktivitätsstruktur des originalen Prozessdefinitionsmodells in eine Schleife einzuschachteln, wie in Abbildung 3.16 vereinfacht dargestellt. Diese zusätzliche Schleife bewirkt, dass die Aktivität A überhaupt erst Nachfolger von Aktivität B während der Prozessdurchführung sein kann. Damit ist es möglich, zur Prozesslaufzeit den Kontrollfluss zu bereits durchgeführten Aktivitäten zurückzulenken. Im Folgenden wird diese spezielle Schleife mit "Dynamic Iteration Decision (DID)" bezeichnet.

Die zusätzliche Schleife allein reicht allerdings nicht aus, da Rücksprünge von B nach A natürlich nicht mit einer zwischenzeitlichen Ausführung aller bezüglich des Prozessablaufs B nachgelagerten Aktivitäten und aller A vorgelagerten Aktivitäten einhergehen sollen. Die Nicht-Ausführung dieser Aktivitäten kann jedoch unter Zuhilfenahme dynamischer Löschungen bewerk-

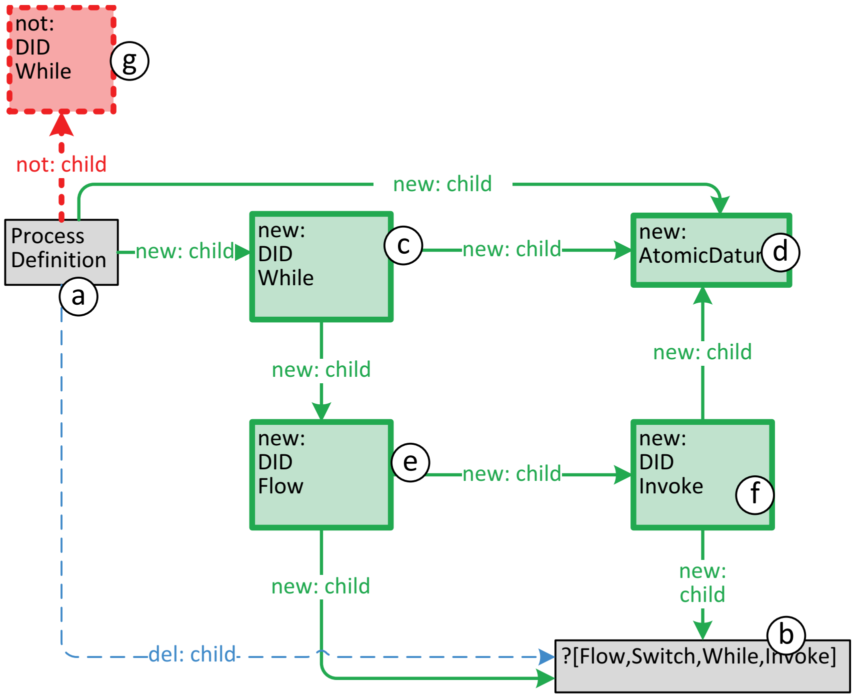


Abbildung 3.17: Graphersetzungsregel addDID zur Anreicherung von “Dynamic Iteration Decisions”

stelligt werden, deren Realisierungen im vorhergehenden Abschnitt erläutert wurden.

Beispiel 3.4 (Anwendung von addDID) Abbildung 3.17 zeigt die Graphersetzungsregel addDID, die eine DID-Schleife in einen bestehenden Prozessgraphen einführt. Diese Regel löst die child-Verbindung vom Prozessdefinitionsknoten ⓐ zur bisher bzgl. Kompositionshierarchie obersten Aktivität ⓑ. Den Platz der obersten Aktivität nimmt die DID-Schleife ⓒ ein. Zusätzlich wird ein neues Datum ⓓ eingeführt, dass das Abbruchkriterium der DID-Schleife darstellt. Das einzige Kind der DID-Schleife ist ein Flow ⓔ mit der bisherigen obersten Aktivität ⓑ und bzgl. Kontrollflussdefinition vorgeschaltetem Invoke ⓕ. Diese Invoke-Aktivität ist nötig, um die Schleifenbedingung direkt nach Betreten der Schleife so zu ändern, dass die Schleife ohne weitere Eingriffe nur genau einmal durchlaufen wird. Die negative Anwendungsbedingung ⓖ verhindert die mehrfache Anwendung von addDID.

In Abbildung 3.18 ist der Prozessgraph nach vollständiger Anreicherung um DRD-Aktivitäten und DID-Aktivitäten dargestellt.

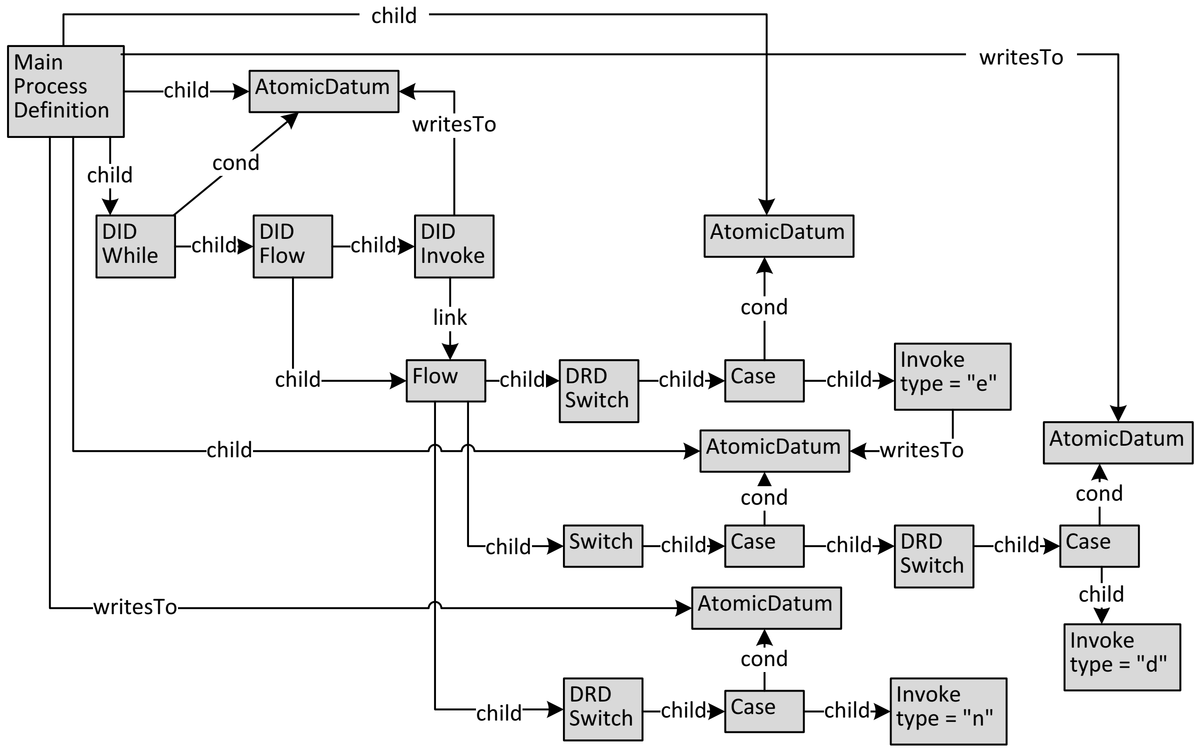


Abbildung 3.18: Prozessgraph nach maximal häufiger Anwendung von add-DRD und addDID

Eingriffe zur Prozesslaufzeit

Als Beispiel soll während der Ausführung der n-Aktivität (notify holder) zur d-Aktivität (disburse compensation) zurückgesprungen werden. Dieser Rücksprung kann bei der Prozessdurchführung notwendig werden, wenn bei der Durchführung von n festgestellt wird, dass die Bankverbindung des Versicherungsnehmers falsch vermerkt ist, die zuvor durchgeführte Schadensauszahlung daher nicht den gewünschten Effekt erzielen kann. Der entsprechende Prozesszustand ist in Abbildung 3.19 dargestellt und ist durch mehrfache Anwendungen von Regeln aus $\mathcal{R}_{bpel}$ erreichbar.

Wiederum ist die Transformation des Prozessdefinitionsmodells so geartet, dass zur Prozesslaufzeit nur Eingriffe vonnöten sind, die Prozessdaten ändern und nicht die Aktivitätsstruktur des Prozessdefinitionsmodells. Somit sind sie auch auf Basis des WebSphere Process Server möglich.

In das Prozessinstanzmodell wird zur Laufzeit folgendermaßen eingegriffen: Zunächst ist die Bedingung der DID-Schleife ⓐ zu ändern, so dass der Kontrollfluss nach Beendigung der Schleife auf dessen Anfang zurückgesetzt wird. Darüber hinaus ist dafür zu sorgen, dass vor- und nachgelagerte Aktivitäten nicht zusätzlich ausgeführt werden. Im Beispiel gibt es in dem Sinne keine nachgelagerten Aktivitäten, jedoch die vorgelagerte e-Aktivität (determine

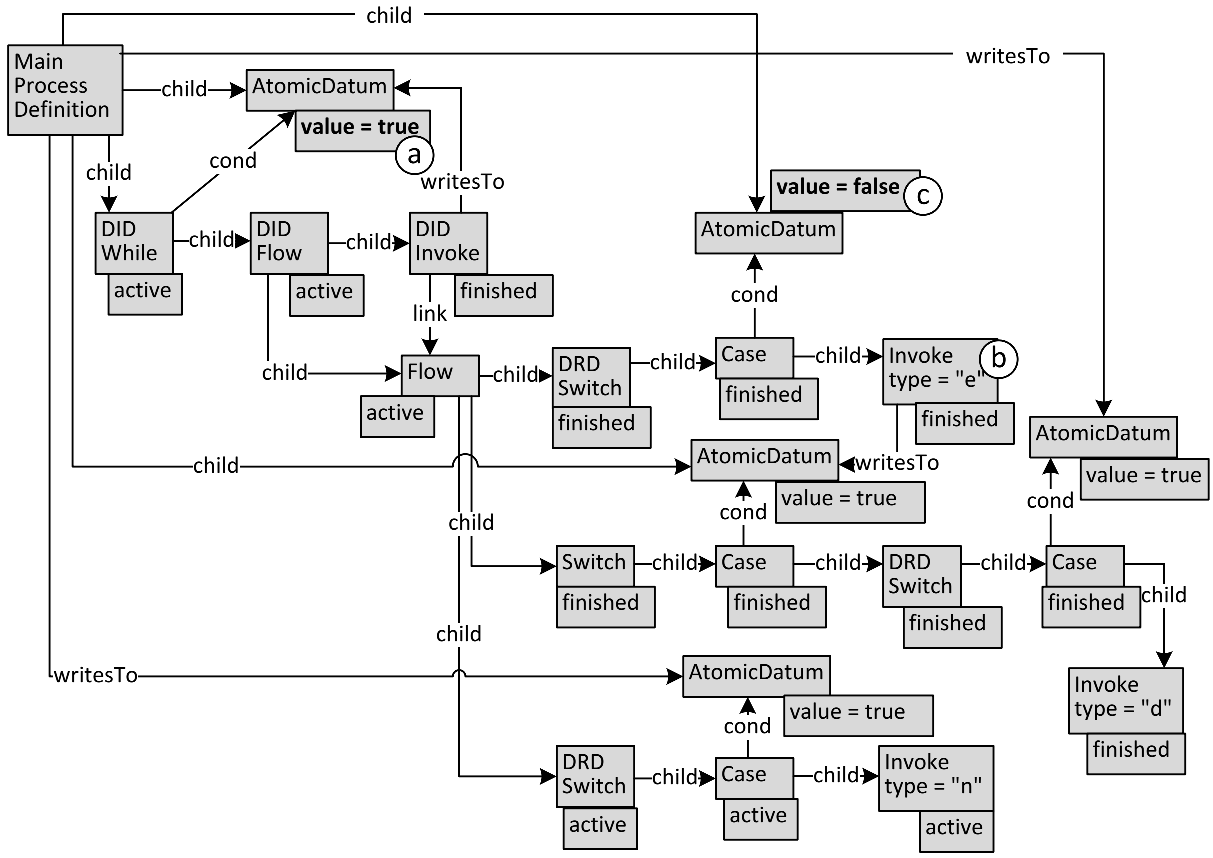


Abbildung 3.19: Bedingungsänderungen im Prozessgraphen zur Simulation eines dynamischen Rücksprungs

eligibility and amount) ⓑ. Daher muss diese aus dem Prozess dynamisch gelöscht werden, was mittels der umgebenen DRD-Aktivität geschieht. Hier wird die Bedingung des einzigen Zweiges auf false gesetzt ⓒ, so dass der Zweig, der die e-Aktivität beinhaltet, bei der wiederholten Iteration nicht betreten wird.

Das Prozessdefinitionsmodell aus Abbildung 3.8 ist insoweit einfach, als dass hier alternative und parallele Ausführungspfade nicht vorkommen. Solche Pfade müssen jedoch bei den Eingriffen zur Laufzeit berücksichtigt werden, d.h. nicht nur vor- und nachgelagerte Aktivitäten müssen berücksichtigt werden, sondern auch parallele und alternative.

Im Folgenden werden Spezialfälle diskutiert, die sich bei Rücksprüngen von bzw. zu parallelen bzw. alternativen Aktivitäten ergeben. Die zugehörigen Abbildungen verwenden eine vereinfachte Variante der konkreten Syntax von Prozessinstanzmodellen (vgl. Abschnitt 2.3), vor deren Anreicherung um dynamikerzeugende Muster. Pfeile in den Modellen deuten Quelle und Ziel des jeweiligen Rücksprungs an.

Rücksprünge von und zu parallel ausgeführten Aktivitäten Die Abbildung 3.20 zeigt vier Prozessinstanzmodelle, in denen jeweils ein Spezialfall für dynamische Rücksprünge diskutiert wird.

alleinige Sequenzen Der einfachste Fall ist der Rücksprung innerhalb einer Sequenz, die zu keiner anderen Aktivität parallel durchgeführt wird, wie in Abbildung 3.20(a) dargestellt. Hier sind vor- und nachgelagerte Aktivitäten einfach zu entfernen wie bereits beschrieben.

innerhalb Sequenz In Abbildung 3.20(b) findet ein Rücksprung innerhalb einer Sequenz statt, die parallel zu einer weiteren Aktivitätssequenz durchgeführt wird. Die Schwierigkeit hierbei liegt darin, dass auch Aktivitäten in nebenläufigen Pfaden von dem Rücksprung betroffen sind. Aufgrund der Ausführungssemantik der Prozessdefinitionsmodelle (vgl. Unterabschnitt 2.5.3) muss im Beispiel auch Aktivität 5 beendet werden, damit die aktuelle Iteration der DID-Schleife beendet und neu begonnen werden kann. Dementsprechend muss über DRD-Verzweigungen in der nächsten Iteration insbesondere Aktivität 4 übersprungen werden.

aus Parallelität Ein Sprung aus einem parallelen Pfad heraus betrifft auch parallele Ausführungspfade. In dem Beispiel aus Abbildung 3.20(c) muss mit Aktivität 3 auch Aktivität 5 beendet werden. Vorher kann der Rücksprung zu Aktivität 1 nicht stattfinden. Gemeinsame Nachfolgeraktivitäten wie Aktivität 6 werden als nachgelagerte Aktivitäten dynamisch entfernt.

in Parallelität Sprünge in parallele Pfade erfordern, dass nur der parallele Pfad ab der Aktivität durchgeführt wird, zu der zurückgesprungen wird. Im Beispiel von Abbildung 3.20(d) ist insbesondere durch dynamische Löschungen zu verhindern, dass die Aktivitäten 4 und 5 nochmals durchgeführt werden.

Rücksprünge zu bedingt ausgeführten Aktivitäten Rücksprünge sind generell zu allen Aktivitäten innerhalb eines Prozessinstanzmodells möglich. Schwierigkeit bei der Realisierung von Rücksprüngen, wie in Unterabschnitt 3.4.3 beschrieben, bereiten jedoch Rücksprünge zu bedingt ausgeführten Aktivitäten. Bedingt ausgeführte Aktivitäten sind innerhalb von Zweigen (`Case`) einer Verzweigung (`Switch`) oder einer Schleife (`While`) modelliert. Es kann der Fall eintreten, dass eine Aktivität A innerhalb einer `Case`- oder `While`-Aktivität modelliert ist, deren Aktivierungsbedingung zum Zeitpunkt des Rücksprungs nicht zutrifft. In diesem Fall kann der Kontrollfluss nicht wie gehabt über DRD-Aktivitäten so gesteuert werden, dass effektiv zu A zurückgesprungen wird. Dies liegt daran, dass DRD-Aktivitäten nur das dynamische

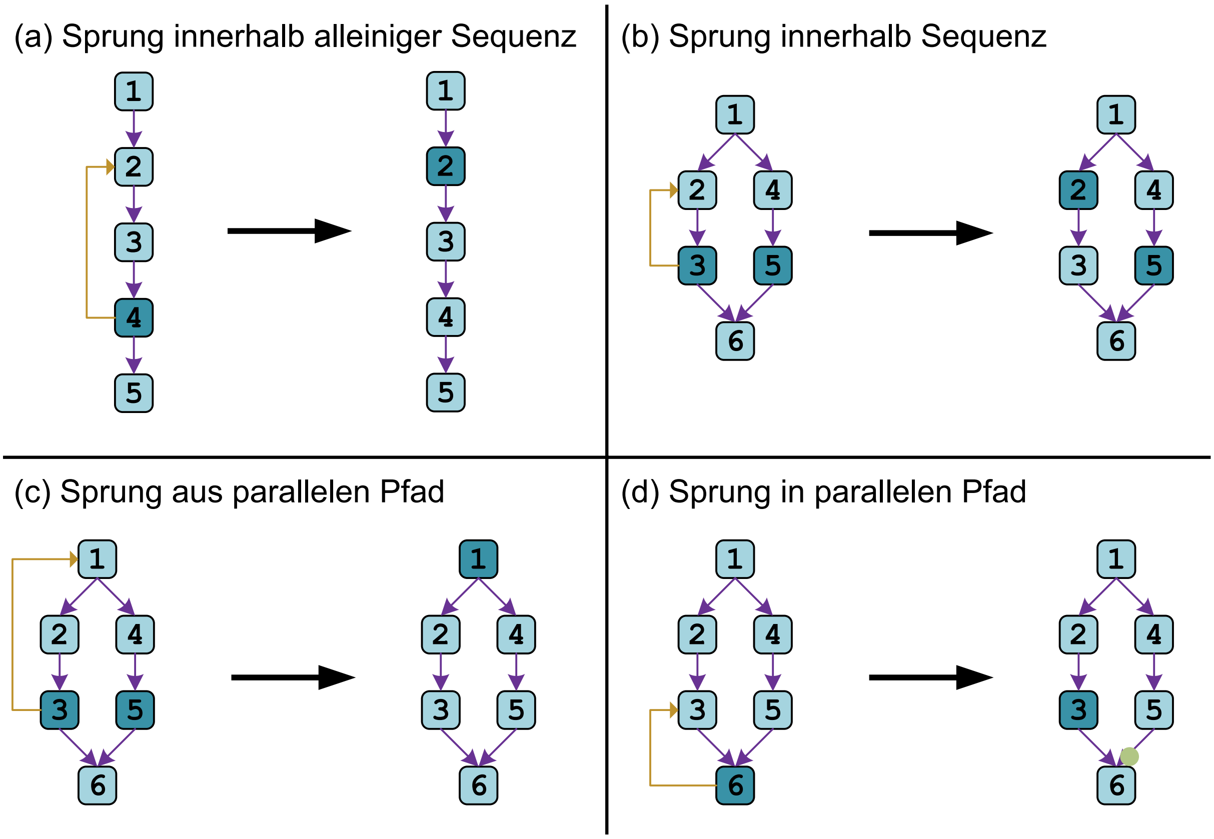


Abbildung 3.20: Sprünge von und zu parallel ausgeführten Aktivitäten [Ehs08, Kapitel 6]

Löschen ermöglichen, jedoch die Ausführung von Aktivitäten in bereits vorab modellierten Zweigen mit unwahrer Bedingung nicht erzwingen können.

Tatsächlich ist für derartige Fälle eine Erweiterung der Transformation der Prozessdefinitionsmodelle erforderlich. Die Aktivierungsbedingungen werden in den WS-BPEL-Prozessdefinitionsmodellen in textueller Form als Attribute der `Case`- bzw. `While`-Aktivitäten formuliert. Die textuellen Ausdrücke stellen auslagenlogische Formeln dar, die ebenfalls transformiert werden müssen. Die Erweiterung dieser Bedingungstransformation ist, eine Bedingung φ in den ursprünglichen Prozessdefinitionsmodellen zu einer Bedingung

$$(\neg X_{overrule} \wedge \varphi) \vee (X_{overrule} \wedge X_{execute})$$

zu transformieren. Hierbei sind $X_{overrule}$ und $X_{execute}$ aussagenlogische Variablen, die die ursprüngliche Bedingung überstimmen können und somit die Ausführung der betreffenden `Case`-Aktivität ggf. erzwingen bzw. verhindern.

Bei der normalen Ausführung der Prozessdefinitionsmodelle sind die Werte der $X_{overrule}$ alle `false`. Dementsprechend hängt es allein von der ursprünglichen Bedingung φ ab, ob die erweiterte Bedingung insgesamt zu wahr evaluiert. Ist eine `Case`-/`While`-Aktivität von einem Rücksprung betroffen, d.h.

befindet sich das Rücksprungziel innerhalb der `Case`-Aktivität, so valuiert das $X_{overrule}$ innerhalb ihrer Bedingung zu `true`. Hierdurch ist der Wert von φ für den Wert der erweiterten Bedingung unerheblich. In diesem Fall ist der Wert der erweiterten Bedingung identisch mit $X_{execute}$.

Die Transformation kann in Graphersetzungsregeln formuliert werden. Die entsprechenden Graphersetzungregeln sind allerdings sehr komplex. Daher wird von einer näheren Erläuterung der betreffenden Regeln an dieser Stelle abgesehen.

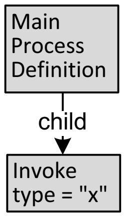
3.4.4 Gesamttransformation

Die Beispiele aus Unterabschnitt 3.4.1 bis Unterabschnitt 3.4.3 sind insoweit vereinfacht, als dass die jeweiligen Graphersetzungsregeln nur isoliert voneinander und im Falle von addDAI und addDRD nur für jeweils eine Anwendungsstelle durchgeführt wurden. Die Graphersetzungsregeln sind jedoch so spezifiziert, dass sie in Kombination miteinander ausgeführt werden können. Eine Regelpriorisierung sorgt dabei dafür, dass zuerst addDAI, dann addDRD und dann addDID jeweils so lang wie möglich ausgeführt wird. DAI-Aktivitäten werden bei der bzw. den Anwendungen von addDRD nicht berücksichtigt, daher die negative Anwendungsbedingung bei @ in addDRD (Abbildung 3.13). In der Graphersetzungsregel addDID muss nur noch die DID-Schleife eingefügt werden; die für die dynamischen Rücksprünge ebenfalls notwendigen Mittel zum Löschen von Aktivitäten sind zuvor durch addDRD generiert worden.

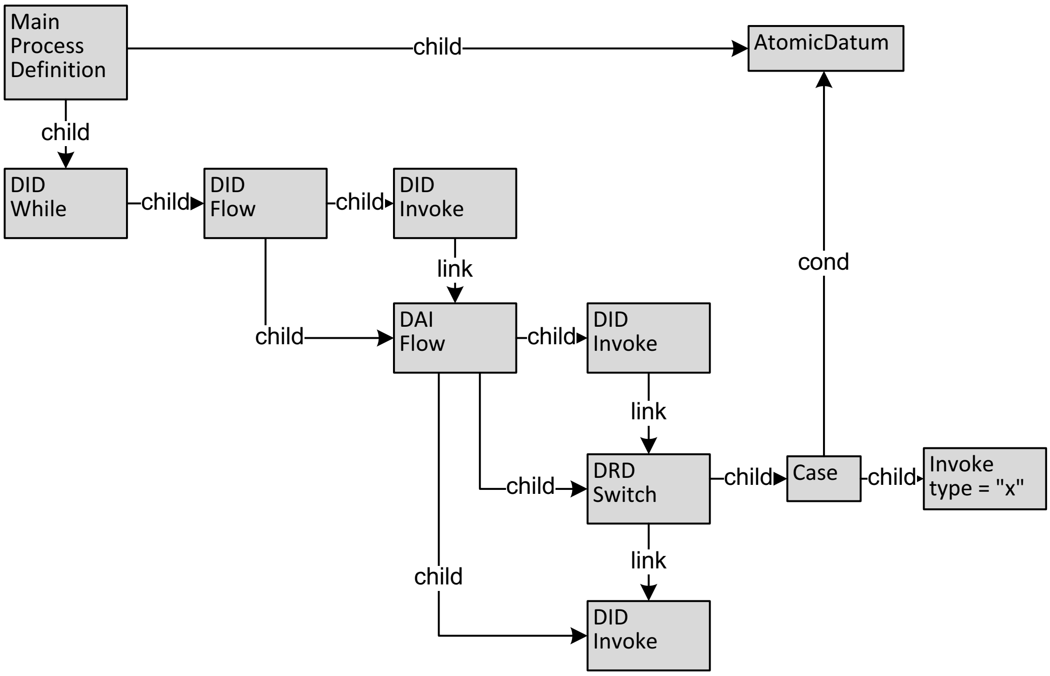
Beispiel 3.5 (Gesamttransformation) In Abbildung 3.21 ist eine Gesamttransformation dargestellt. Hierbei zeigt Abbildung 3.21(a) die Ausgangssituation in Form eines sehr einfachen Prozessgraphen mit genau einer Invoke-Aktivität x. Abbildung 3.21(b) stellt die Situation nach vollständiger Anwendung der Regeln aus $\mathcal{R}_{trans}$ dar. Hier zeigt sich, dass die Transformation so gestaltet ist, dass DAI-Aktivitäten außerhalb der DRD-Verzweigung verbleiben. Andernfalls würden vor x dynamisch eingefügte Aktivitäten nicht ausgeführt, falls x anschließend dynamisch gelöscht würde.

3.4.5 Standardverhalten dynamikerzeugender Muster

Das standardmäßige Verhalten eines angereicherten Prozessdefinitionsmodells Xa entspricht dem des originalen Prozessdefinitionsmodells X. Das bedeutet, dass ohne die angesprochenen dynamischen Eingriffe in Xa genau



(a) Einfacher Prozessgraph vor Gesamttransformation



(b) Einfacher Prozessgraph nach Gesamttransformation

Abbildung 3.21: Gesamttransformation auf einem einfachen Prozessgraphen

die Abläufe möglich sind, die bereits in X möglich waren. Das ist so gewollt, da dynamische Änderungen dazu dienen sollen, unvorhergesehene Anforderungen an den Prozess abzubilden, nicht jedoch, um vormodellierten Abläufe a-priori zu ändern.

Dass sich das Verhalten beider Prozessdefinitionsmodelle gleicht, ist durch das Verhalten der dynamikerzeugenden Muster begründet.

DAI-Aktivitäten DAI-Aktivitäten sind standardmäßig effektfrei; ohne Eingriff zur Prozesslaufzeit werden sie durchgeführt, ohne dabei andere Prozessinstanzen zu erzeugen.

DRD-Aktivitäten Der einzige Zweig in DRD-Aktivitäten wird ohne Eingriff zur Prozesslaufzeit durchgeführt. Die neu eingeführte Prozessvariable für den Zweig ist also im Standardfall true.

DID-Aktivitäten Die Herstellung des originalen Verhaltens des angereicherten

Prozessdefinitionsmodells Xa ist im Falle der dynamischen Rücksprünge etwas aufwändiger. Hier ist dafür zu sorgen, dass die Schleife im Standardfall genau einmal betreten wird. Dies wird dadurch erreicht, indem die erste Invoke-Aktivität in der Schleife eine neu generierte Aktivität ist, die die Schleifenbedingung direkt auf false setzt.

3.5 Implementierung auf dem WebSphere Process Server

Im vorhergehenden Abschnitt 3.3 wurden die zur Realisierung der dynamikerzeugenden Muster notwendigen Änderungen des Prozessdefinitionsmodells und Eingriffe zur Prozesslaufzeit anhand einer Regelmenge $\mathcal{R}_{trans}$ und beispielhaften Prozessgraphen erläutert. Die dort dargestellte Vorgehensweisen lassen sich in die Dynamik-Schicht aus Abbildung 3.4 übertragen. Die Übertragung ist nicht trivial, da die Technik des WebSphere Process Servers, insbesondere die Speicherung von und der Zugriff auf Prozessdefinitionsmodelle und Prozessinstanzen innerhalb des WebSphere Process Servers, kompliziert ist. Des Weiteren sind im vorhergehenden Abschnitt einige Probleme ausgeblendet worden, die nicht von der Übertragung auf den WebSphere Process Server herrühren. Dazu gehört beispielsweise die Behandlung von Datenflüssen bei dynamischen Änderungen.

3.5.1 WS-BPEL-Transformator

Prozessdefinitionsmodelle in WS-BPEL werden in XML serialisiert. Ein XML-Schema für WS-BPEL bestimmt dabei die Teilsprache von XML aller gültigen serialisierten WS-BPEL-Prozessdefinitionsmodelle. GROOVE-Graphen werden in der Graph Exchange Language (GXL) [WKR01] gespeichert, die ebenfalls als Teilsprache von XML mittels eines XML-Schemas definiert ist.

Zwei Alternativen zur Realisierung des WS-BPEL-Transformators, der innerhalb der Dynamik-Schicht (vgl. Abbildung 3.4) Prozessdefinitionsmodelle in WS-BPEL um dynamikerzeugende Muster anreichert, sind naheliegend:

indirekte Transformation mittels $\mathcal{R}_{trans}$ Die Verwendung einer Graphgrammatik $(\mathcal{R}_{trans}, \mathrm{pg}(X))$ ist eine Möglichkeit, die Anreicherung eines Prozessdefinitionsmodells X zu realisieren. Der einzige Endzustand im Graphtransitionssystem $\mathfrak{T}_{\mathcal{R}_{bpel},\mathrm{pg}(X)}$ wäre dann gerade $\mathrm{pg}(Xa)$. Durch eine Umkehrabbildung pg^{-1} ließe sich das angereicherte Prozessdefinitionsmodell Xa gewinnen. Insgesamt sind also drei Schritte notwendig, die in Abbildung 3.22 als strichlierte Pfeile angedeutet sind. Der Vorteil dieses Vorgehens ist, dass die Transformation, im Zusammenspiel mit $\mathcal{R}_{bpel}$ bereits

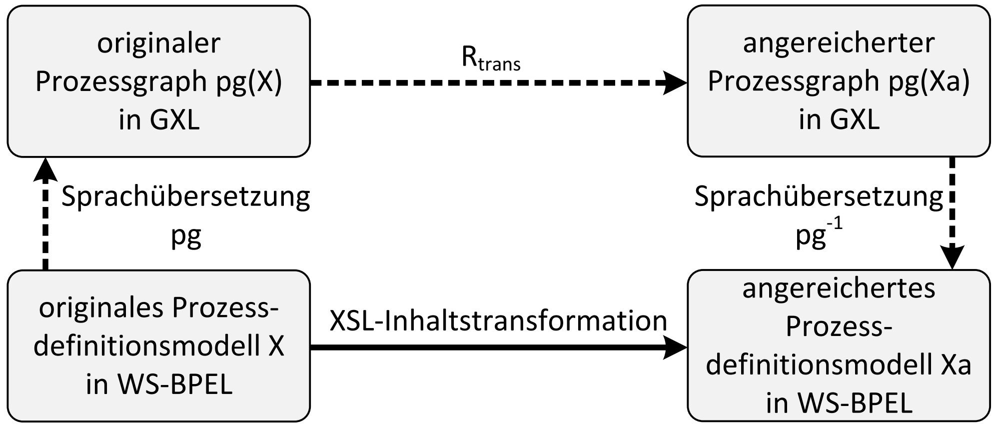


Abbildung 3.22: Strategien für die WS-BPEL-Anreicherung um dynamikerzeugende Muster

erprobt ist; die Korrektheit der Transformationen lassen sich in GROOVE leicht nachprüfen. Der Nachteil ist, dass zwischen den XML-Sprachen WS-BPEL und GXL in beiden Richtungen übersetzt werden muss und dabei insbesondere auch solche Teile, die für die Anreicherung irrelevant sind, beispielsweise Schnittstellendefinitionen oder Parametertypdeklarationen (vgl. Unterabschnitt 2.1.5).

direkte Transformation Die Alternative hierzu ist der direkte Weg vom originalen Prozessdefinitionsmodell X zum angereicherten Prozessdefinitionsmodell Xa ohne weitere Zwischensprachen. Nachteil dieser Vorgehensweise ist, dass $\mathcal{R}_{trans}$ nur noch als konzeptuelle Vorlage einer anderweitig formulierten Transformationsspezifikation verwendet werden kann; der Vorteil ist, dass die zusätzlichen Sprachübersetzungen wegfallen und der zusätzliche Aufwand der Realisierung dieser Übersetzungen nicht betrieben werden muss. Dieser direkte Weg ist in Abbildung 3.22 als durchgezogener Pfeil dargestellt.

In dieser Arbeit wurde der zweite Ansatz gewählt, da die notwendigen Sprachübersetzungen zwischen WS-BPEL und GXL vermutlich wesentlich aufwändiger gewesen wären als die Umformulierung der Graphersetzungsregeln.

Der WS-BPEL-Transformator wurde in der Extensible Stylesheet Language (XSL) implementiert. XSL ist eine Transformationssprache, deren Ausdrücke, so genannte XSL-Templates, auf XML-Dokumente (Transformationsquelle) angewendet werden können und die selbst eine Teilsprache von XML ist. Transformationsziele können (a) Nicht-XML-Dokumente, z.B. LaTeX-Dokumente, (b) XML-Dokumente aus anderen XML-Teilsprachen, also mit anderem XML-Schema, oder (c) XML-Dokumente aus der gleichen XML-Teilsprache sein. Der Fall (b) ist der übliche Fall und wird beispielsweise dann angewendet,

addDAI (dynamisches Einfügen)

```
Wf.bpel

<process name="Wf" ...>
…
<sequence>
  <receive ... >…</receive>
  <invoke name="A">…</invoke>
  <invoke name="B">…</invoke>
  <reply ... >…</reply>
</sequence>
</process>
```

Transformation

```
Wf.dyn.bpel

<process name="Wf.dyn" ...>

…
<sequence>
  <receive ... >…</receive>
  <flow ...>
    <invoke name="DAI">…</invoke>
    <invoke name="A">…</invoke>
    <invoke name="DAI">…</invoke>
  </flow>
  <flow ...>
    <invoke name="DAI">…</invoke>
    <invoke name="B">…</invoke>
    <invoke name="DAI">…</invoke>
  </flow>
  <reply ... >…</reply>
</sequence>
</process>
```

addDRD (dynamisches Löschen)

```
Wf.bpel

<process name="Wf" ...>
…
<sequence>
  <receive ... >…</receive>
  <invoke name="A">…</invoke>
  <invoke name="B">…</invoke>
  <reply ... >…</reply>
</sequence>
</process>
```

Transformation

```
Wf.dyn.bpel

<process name="Wf.dyn" ...>
…
<variables>
  <variable name="skip"
            type="xsd:boolean" />
</variables>
<sequence>
  <receive ... >…</receive>
  <invoke name="drdExecute">
   <output>
     <parameter variable="skip"/>
   <output/><invoke/>
  <switch name="DRD" >
    <case>
      <condition>
        not(getVariableData("skip"))
      </condition>
      <invoke name="A">…</invoke>
    <case/>
  </switch>
  <invoke name="drdExecute">
   <output>
     <parameter variable="skip"/>
   <output/><invoke/>
  <switch name="DRD" >
    <case>
      <condition>
        not(getVariableData("skip"))
      </condition>
      <invoke name="B">…</invoke>
    <case/>
  </switch>
  <reply ... >…</reply>
</sequence>
</process>
```

addDID (dynamische Rücksprünge)

```
Wf.bpel

<process name="Wf" ...>
…
<sequence>
  <receive ... >…</receive>
  <invoke name="A">…</invoke>
  <invoke name="B">…</invoke>
  <reply ... >…</reply>
</sequence>
</process>
```

Transformation

```
Wf.dyn.bpel

<process name="Wf.dyn" ...>
…
<variables>
  <variable
            name="iterate"
            type="xsd:boolean" />
</variables>
<sequence>
 <receive ... >…</receive>
<while name="DID" >
  <condition>
    getVariableData("iterate")
  </condition>
  <sequence>
    ...
    <invoke name="A">...</invoke>
    ...
    <invoke name="B">...</invoke>
    ...
  </sequence>
</while>
<reply ... >…</reply>
</sequence>
</process>
```

Abbildung 3.23: XSL-Transformationen auf Serialisierungen von WS-BPEL-Prozessdefinitionsmodellen (nach [Ehs08, Abbildung 4.2])

wenn in XML serialisierte Daten zur Präsentation in eine Auszeichnungssprache wie XHTML überführt werden müssen. Der WS-BPEL-Transformator fällt jedoch in die Kategorie (c): Quell- und Zieldokument sind jeweils gültige WS-BPEL-Serialisierungen.

Hauptbestandteil von Graphgrammatiken sind Graphersetzungsregeln, die auf Graphen angewendet werden. In XSL-Spezifikationen werden analog dazu XSL-Templates definiert, die auf XML-Dokumente angewendet werden. Die XSL-Spezifikation des WS-BPEL-Transformators zerfällt in drei Gruppen von XSL-Templates. Jede Gruppe realisiert dabei die Anreicherung des originalen WS-BPEL-Prozessdefinitionsmodells um ein dynamikerzeugendes Muster (vgl. Abschnitt 3.3).

Abbildung 3.23 zeigt die Wirkung der – wiederum der Übersichtlichkeit halber hier isoliert voneinander – angewendeten XSL-Template-Gruppen auf ein Prozessdefinitionsmodell `Wf.bpel`. Das Ergebnis ist ein angereichertes Prozessdefinitionsmodell `Wf.dyn.bpel`. Das Prozessdefinitionsmodell besteht dabei neben den obligatorischen `receive`- und `reply`-Aktivitäten aus nur zwei `invoke`-Aktivitäten namens `A` und `B`. Die Reihenfolge der Vorkommen der Aktivitäten innerhalb der komplexen `sequence`-Aktivität gibt die Ausführungsreihenfolge der Aktivitäten an. An dieser Stelle hätte auch eine Flow-Aktivität verwendet werden können, innerhalb derer die Aktivitäten linear mittels Links verkettet wären. In der textuellen XML-Serialisierung führt der Gebrauch einer `sequence`-Aktivität jedoch zu einer kompakteren Formulierung. Die XSL-Template-Gruppen transformieren das originale Prozessdefinitionsmodell nun wie folgt:

addDAI Jede `Invoke`-Aktivität wird ersetzt durch eine DAI-`Flow`-Aktivität und in diese eingeschachtelt zusammen mit zwei neuen mittels `Links` vor- und nachgelagerten DAI-`Invoke`-Aktivitäten.

addDRD Dynamikerzeugende Muster für dynamisches Löschen werden analog zur Graphersetzungsregel aus Abbildung 3.13 in das WS-BPEL-Prozessdefinitionsmodell eingeführt. Aus technischen Gründen muss vor jeder DRD-`switch`-Aktivität hier jedoch zusätzlich das Datum für die Zweigbedingung in die Prozessvariable skip kopiert werden. Von diesem technischen Zwang wurde bei den Überlegungen in der Graphgrammatik $\mathcal{R}_{trans}$ noch abstrahiert. Er ändert aber nichts Grundsätzliches an der Transformation.

addDID Wie in der Graphersetzungsregel von Abbildung 3.17 wird hier die Ablaufstruktur in eine umgebende `While`-Aktivität gefasst. Receive- und reply-Aktivitäten werden nicht mit eingeschlossen, da diese immer nur sinnvollerweise einmal beim Prozessstart bzw. -ende durchgeführt werden.

Auf die Details der XSL-Templates wird nicht weiter eingegangen, da sie im Wesentlichen nur eine Umformulierung der Graphersetzungsregeln aus Abschnitt 3.4 sind. Stattdessen sei bzgl. der Details auf [Ehs08, Kapitel 5] verwiesen.

3.5.2 Dynamik-Komponente

Die Anreicherung des WS-BPEL-Prozessdefinitionsmodells um dynamikerzeugende Muster ist nur eine Vorbereitung, um dynamische Änderungen zur Prozesslaufzeit auf technisch mögliche Eingriffe abzubilden. Technisch möglich sind im WebSphere Process Server nur Manipulationen an den Instanzdaten, nicht jedoch an der Ablaufstruktur. Hierbei liegen im WebSphere Process Server Prozessdefinitionsmodelle und -instanzen technisch getrennt als Dateien bzw. Verbundobjekte im Hauptspeicher vor (vgl. Unterabschnitt 3.1.4) – und nicht vereinigt wie in Prozessgraphen aus Abschnitt 3.4. Erschwerend kommt hinzu, dass der WebSphere Process Server nicht beliebige Manipulationen an den Instanzdaten zulässt. Dieser Sachverhalt erfordert, dass Instanzdaten wiederum aufgeteilt werden in solche, die sich auf Aktivitäten und Prozessdaten des ursprünglichen Prozessdefinitionsmodells beziehen und solche, auf die zwecks Auslösung einer dynamischen Änderung vom Prozessbeteiligten Einfluss genommen werden muss.

Prozessinstanzdaten im WebSphere Process Server werden daher um erweiterte Prozessinstanzdaten ergänzt. Diese werden separat von den Prozessinstanzdaten innerhalb eines Datenabstraktionsmoduls der Dynamik-Komponente aus Abbildung 3.4 vorgehalten. Die Anreicherung in der Dynamik-Schicht ist also eine zweifache: Zur Definitionszeit werden Prozessdefinitionsmodelle in WS-BPEL um dynamikerzeugende Muster angereichert, zur Laufzeit die eigentlichen Prozessinstanzdaten um erweiterte Prozessinstanzdaten.

Die erweiterten Prozessinstanzdaten stellen folgende Informationen dar:

DAI-Bindungen DAI-Aktivitäten rufen stets die Funktion `daiExecute(String daiID)` der Dynamik-Komponente auf, wenn Sie durchgeführt werden. Im Normalfall bewirkt diese Funktion nichts. Ihre Implementierung stützt sich auf eine Tabelle ab, die jeder DAI-Aktivität entweder ein Prozessdefinitionsmodell zuordnet oder – im Normalfall – einen noop[1]-Wert. Ist für eine DAI-Aktivität ersteres der Fall, so entspricht diese Zuordnung einer invokes-Kante im Prozessgraphen wie in Abbildung 3.12. Die Tabelle speichert somit die Bindungsinformationen für die DAI-Aktivitäten, über die ein dynamisches Einfügen umgesetzt wird. Komplementär zu Funktion `daiExecute`, die Bestandteil der Schnittstelle zwischen WebSphere Process Server und Dynamik-Komponente (WPS-DK-Schnittstelle) ist, gibt es eine Funktion `setDAI(String daiID, String pdID)` in der Schnittstelle zwischen Dynamik-Komponente und Prozessmodelleditor (PME-DK-Schnittstelle). Über diese können die Bindungsinformation (indirekt mittels eines Prozessmodelleditors) dann manipuliert werden, wenn der Prozessbeteiligte eine dynamische Einfügeoperation auslöst.

DRD-Zuweisungen DRD-Zweig-Aktivitäten werden zur Laufzeit nur ausgeführt, wenn die betreffende Zweigbedingung zutrifft. Diese Bedingung ist von einem Wahrheitswert abhängig, der ebenfalls spezifisch für jeden DRD-Zweig in der Dynamik-Komponente vorgehalten wird. Vor Ausführung einer

[1] no operation

DRD-Switch-Aktivität wird der betreffende Wahrheitswert mittels Aufruf der WPS-DK-Schnittstellenfunktion `didExecute(String drdID)` in die vom WS-BPEL-Transformator generierte Prozessvariable `skip` kopiert (vgl. Abbildung 3.23), die schließlich unmittelbar vor Ausführung des DRD-Zweiges ausgewertet wird. Über die Funktion `setDRD(String drdID)` der PME-DK-Schnittstelle können die Zuordnungen von DRD-Zweig zu Wahrheitswert zur Laufzeit geändert werden. Standardmäßig ist jeder DRD-Zweig dem Wert `true` zugeordnet. Der Wert false führt zu einer Nichtausführung des Zweiges und der darin enthaltenen ursprünglichen Aktivität und simuliert somit eine dynamische Löschung.

DID-Zuweisungen Die Bedingung für DID-While-Aktivitäten wird analog zu den Bedingungen der DRD-Zweige behandelt. Auch hier gibt es eine Funktion `didExecute()` zum Abfragen und `setDID()` zum Setzen der Schleifenvariablen.

3.5.3 Datenflüsse

Datenflüsse in WS-BPEL-Prozessdefinitionsmodellen werden implizit modelliert. Sie ergeben sich durch Zugriffe der Aktivitäten auf die Prozessvariablen. Ein lesender bzw. schreibender Zugriff auf eine Prozessvariable V durch eine Aktivität A ist dann gegeben, wenn V als aktualer Eingabe- bzw. Ausgabeparameter in der durch A realisierten Schnittstelle vorkommt. Wird V als aktualer Ausgabeparameter von Aktivität A und als aktualer Eingabeparameter von Aktivität B verwendet, so existiert ein Datenfluss von A nach B.

Datenflussaspekte spielen insbesondere beim dynamischen Einfügen einer Aktivität A in einen Prozess P insofern eine Rolle, als dass für die Formalparameter von A geeignete Prozessvariablen aus P vom Prozessbeteiligten ausgewählt werden müssen, sofern A Formalparameter besitzt. Die Notwendigkeit hierfür ist nicht ansatzbedingt; sie ergäbe sich auch, wenn die Aktivitätsstruktur in Prozessdefinitionsmodellen zur Laufzeit direkt geändert wird anstatt das Einfügen durch den zuvor beschriebenen Ansatz zu simulieren.

3.6 Benutzersichten

Der Ansatz zur Simulation von Dynamik ist so realisiert worden, dass die zusätzliche Komplexität, die durch die Anreicherung der Prozessdefinitionsmodelle und -instanzdaten entsteht, vor Prozessbeteiligten verborgen werden kann. Insbesondere werden vor Prozessbeteiligten die dynamikerzeugenden Muster verborgen sowie die Tatsache, dass dynamisch eingefügte Aktivitäten im WebSphere Process Server in separaten Prozessinstanzen durchgeführt werden. Dies wird durch Prozessinstanzmodelle erreicht, die lokal auf dem Rechner des Prozessbeteiligten im Prozessmodelleditor vorgehalten werden und eine Benutzersicht auf die Informationen innerhalb des WebSphere Process Servers und der Dynamik-Komponente darstellen.

Beispiel 3.6 (Lokales Prozessinstanzmodell) In Abbildung 3.24 ist der Zusammenhang zwischen einem lokalen Prozessinstanzmodell (oben), das im Prozessmodelleditor angezeigt und editiert wird, der Dynamik-Komponente (Mitte) und dem WebSphere Process Server (unten) beispielhaft dargestellt.

1. Zunächst müssen die lokalen Prozessinstanzmodelle erzeugt werden. Im Beispiel ist der Prozess bereits bis determine eligibility and amount fortgeschritten. Die automatische Erzeugung geschieht per Kommandoaufruf im Prozessmodelleditor.

2. Am lokalen Prozessinstanzmodell können nun dynamische Änderungen werden, d.h. die Aktivitätsstruktur im lokalen Modell wird direkt geändert. In diesem Fall wird hinter der gerade aktiven Aktivität eine Aktivität determine payee eingefügt.

3. Die Änderung wird an die Dynamik-Komponente übermittelt und wird dort so umgesetzt, wie in den Abschnitten zuvor erläutert.

4. Der Prozessbeteiligte kann nun den Prozess weiterführen. In der Situation rechts in Abbildung 3.24 ist gerade die dynamisch eingefügte Aktivität determine payee aktiv.

5. Durch erneutes Erzeugen des lokalen Prozessinstanzmodells werden die geänderten Fortschrittsinformationen auch im lokalen Prozessinstanzmodell angezeigt.

Die lokalen Prozessinstanzmodelle ermöglichen den Prozessbeteiligten ein vom WebSphere Process Server entkoppeltes Arbeiten, d.h. dynamische Änderungen werden nicht sofort aktiv. Das ist nötig, da nicht alle dynamische Änderungen aus technischer und fachlicher Sicht sinnvoll sind. Nicht sinnvolle Änderungen sollten nicht an die Dynamik-Komponente und den WebSphere Process Server propagiert werden. In Kapitel 4 ist erläutert, wie vor der Änderungspropagation in Schritt ③ technische und fachliche Prüfungen in dynamisch geänderten Prozessinstanzmodellen vorgenommen werden können.

In Abbildung 3.25 ist ein Screenshot des Prozessmodelleditors abgebildet.

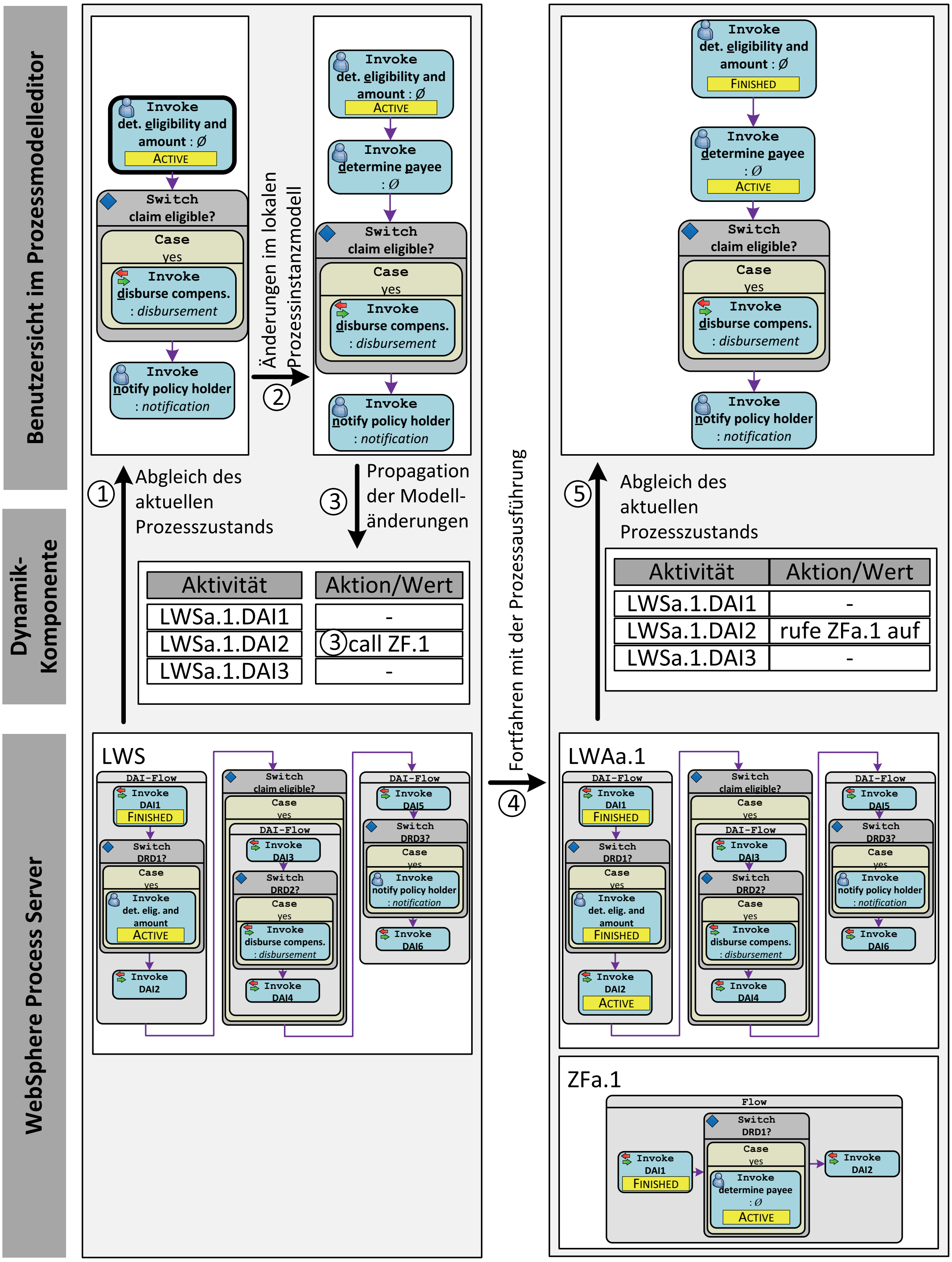


Abbildung 3.24: Abgleich eines Prozessinstanzmodells im Prozessmodelleditor mit den Prozessinstanzen im WebSphere Process Server

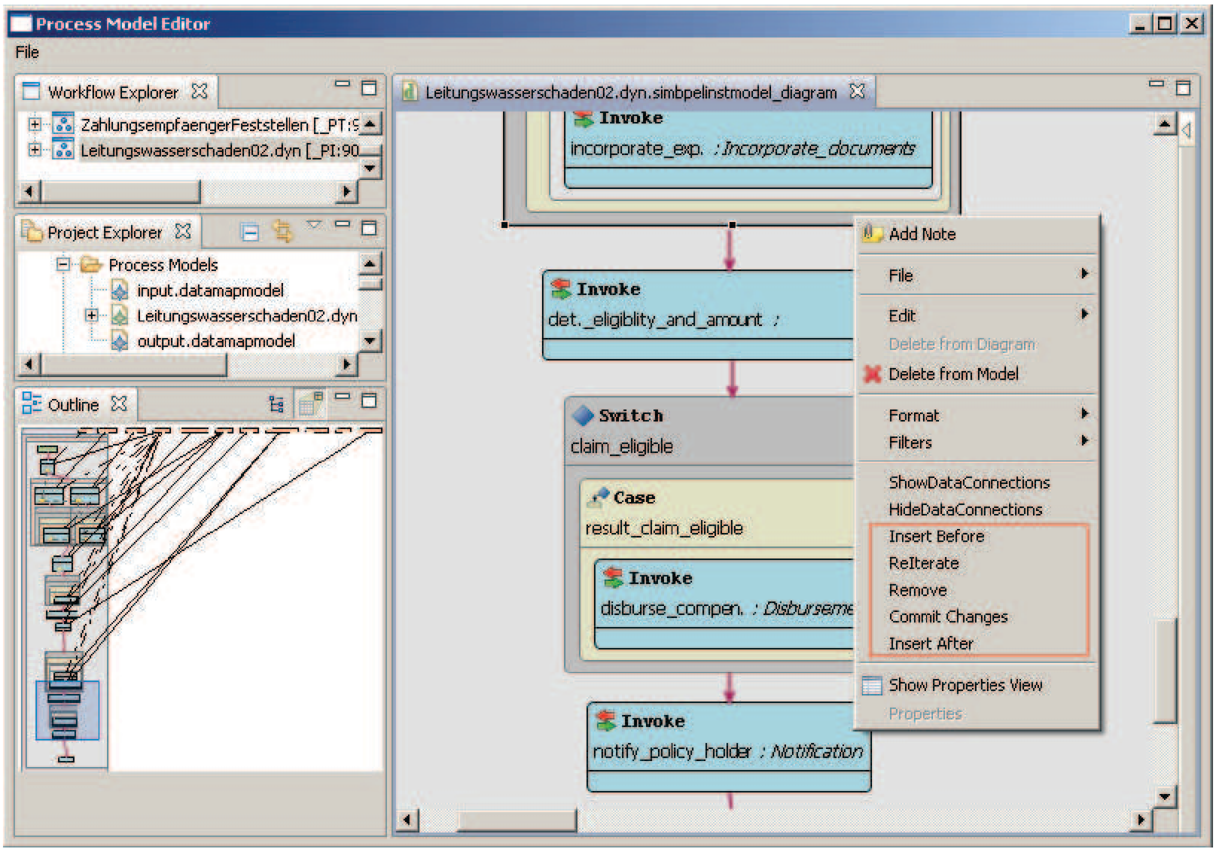


Abbildung 3.25: Prozessmodelleditor zur Anzeige und Manipulation von Prozessinstanzmodellen

Der Prozessmodelleditor zeigt Prozessinstanzmodelle in der Syntax an, die in Abschnitt 2.3 erläutert wurde. In diesen Prozessinstanzmodellen können dynamische Änderungen über folgende Kommandos vorgenommen werden (eingerahmt in der Abbildung):

Insert After zum dynamischen Einfügen einer Aktivität hinter einer bereits vorhandenen. Hierbei ist die Liste links oben eine Auflistung aktuell vorhandener Prozessdefinitionsmodelle. Die Selektion einer dieser Prozessdefinitionsmodelle, beispielsweise determine payee (ZahlungsempfaegerFeststellen), legt fest, dass die darin definierte (komplexe) Aktivität dynamisch in das Prozessinstanzmodell rechts einzufügen ist. Die Einfügestelle wird mittels Selektion einer der Aktivitäten im Diagramm rechts bestimmt, beispielsweise determine eligibility and amount.

Insert Before zum dynamischen Einfügen einer Aktivität vor eine bereits vorhandene. InsertBefore ist ansonsten wie InsertAfter zu verwenden.

Remove zum dynamischen Löschen einer Aktivität. Zum Auslösen dieses Kommandos ist die Selektion der dynamisch zu löschenden Aktivität rechts im Prozessinstanzmodell nötig.

ReIterate für dynamische Rücksprünge. Dieses Kommando kann ausgelöst werden, wenn zunächst die Quelle und dann das Ziel rechts im Prozessinstanzmodell selektiert wird.

Dynamische Änderungen im lokalen Prozessinstanzmodell können mit dem Kommando Commit Changes an die Dynamik-Komponente propagiert werden. In umgekehrter Richtung können über das Kommando Get Local Model (nicht abgebildet) lokale Prozessinstanzmodelle erzeugt werden. Diese werden insoweit inkrementell erzeugt, als dass ab der zweiten Erzeugung die bisherigen Layoutinformationen des Prozessinstanzmodell-Diagramms erhalten bleiben.

3.7 Diskussion

Die Erkenntnis ist nicht neu, dass statische Prozessmanagementsysteme für die Unterstützung dynamischer Prozesse ungeeignet sind. Es existiert eine Reihe von Arbeiten, die sich mit genau dieser Problematik auseinandersetzen [WRR07a]. Auf Einzelheiten und Unterschiede dieser Arbeiten zu dynamischen Prozessmanagementsystemen wird in Abschnitt 3.8 eingegangen. Gemein ist den bisherigen Arbeiten jedoch, dass die erarbeiteten Konzepte und Implementierungen vorhandene Prozessmanagementsysteme unberücksichtigt lassen. Bei den entwickelten Forschungsprototypen lassen sich mehrere Arten unterscheiden, die im Folgenden diskutiert werden.

3.7.1 Neuentwicklungen

Alle dem Autor bekannten Forschungsprotoypen von dynamischen Prozessmanagementsystemen sind Neuentwicklungen, die unabhängig von bestehenden (kommerziellen) Prozessmanagementsystemen entstanden sind. Die Anforderung, auch dynamische Prozesse durch den Prototypen zu unterstützen war bei diesen Systemen *a-priori* von zentraler Bedeutung. Die *Vorteile*, ein dynamisches Prozessmanagementsystem von Grund auf neu zu entwickeln, sind offenkundig:

Verringerter Einarbeitungsaufwand Die Einarbeitung in ein bestehendes System, insbesondere in dessen Programmierschnittstellen, nimmt Zeit in Anspruch. Diese Zeit kann bei einer Neuentwicklung gespart werden.

Kürzere Turn-Around-Zeiten Die Komplexität von gänzlich neu entwickelten Forschungsprototypen ist im Allgemeinen deutlich geringer als die von bestehenden, kommerziellen Prozessmanagementsystemen. Das beschleunigt die Entwicklung insoweit, als dass diese Forschungsprototypen normalerweise viel schneller geladen und daher besser getestet werden können.

Konzeptuelle Freiheit Der aus wissenschaftlicher Sicht wichtigste Grund für eine Neuentwicklung ist, dass nur hierdurch vollständige konzeptuelle Freiheit gewährt werden kann. Wird hingegen ein bestehendes Prozessmanagementsystem erweitert, wirken sich dessen Realisierungskonzepte zwangsweise auch auf Erweiterungen aus.

Bei den Neuentwicklungen lassen sich wiederum zwei Arten voneinander trennen:

Grundsätzliche Neuentwicklungen Zu den grundsätzlichen Neuentwicklungen zählen alle Prototypen, die von Grund auf neu mit herkömmlichen Implementierungsmitteln realisiert wurden. Ein Vertreter dieser Art ist das ADEPT-System [RD98, Rei00]. Hierbei wird unter "herkömmlichen" Mitteln eine imperative oder objektorientierte Implementierungssprache wie C, C++ oder Java verstanden. Diese Arbeiten verzichten auf die Verwendung fortschrittlicher aber auch experimenteller Implementierungs- bzw. Spezifizierungswerkzeuge. Hierdurch bleibt die volle konzeptuelle Freiheit erhalten sowie Abhängigkeit von bestimmten Entwicklungswerkzeugen und Ausführungsumgebungen gering.

Neuentwicklungen auf Basis von fortschrittlichen Werkzeugen Hierzu zählen Prototypen, die ebenfalls von Grund auf neu entwickelt wurden allerdings mit fortschrittlichen und experimentellen Werkzeugen. Das in Abschnitt 2.7 beschriebene AHEAD-System ist ein Vertreter dieser Art. AHEAD liegt zwar ebenfalls kein zuvor vorhandenes, durch AHEAD erweitertes Prozessmanagementsystem zugrunde, jedoch wurden bei der Realisierung von AHEAD fortschrittliche, graphbasierte Werkzeuge wie PROGRES und UPGRADE eingesetzt. Hieraus geben sich Vorteile bzgl. der Verständlichkeit und Änderbarkeit der Funktionalität des AHEAD-Prototyps. Die Nachteile gegenüber der Beschränkung auf herkömmliche Implementierungswerkzeuge sind die technische Abhängigkeit von weiteren Forschungsprototypen und die damit einhergehende Verschlechterung der Portabilität.

3.7.2 Verwendung bestehender Prozessmanagementsysteme

Auch die *a-posteriori* durchgeführt Änderung bzw. Erweiterung eines bestehenden, statischen Prozessmanagementsystems ist eine Alternative zur

Realisierung eines dynamischen Prozessmanagementsystems. Die im vorhergehenden Unterabschnitt 3.7.1 aufgeführten Vorteile sind natürlich Nachteile dieser Vorgehensweise und umgekehrt. Die *Vorteile* der Verwendung bestehender Systeme sind die Folgenden:

Wiederverwendung von Standardfunktionalität Bestehende Systeme besitzen gewisse Grundfunktionen, die in typischen Forschungsprototypen aufgrund eines hohen Implementierungsaufwands ohne wissenschaftlichen Reiz fehlen oder unvollständig implementiert sind. Typische Grundfunktionen sind beispielsweise die von benutzerfreundlichen Worklists (Posteingangskörben) oder Administrationsoberflächen für die technische Verwaltung von Prozessinstanzen und -definitionen. Ebenso fällt die Anbindung bestehender Prozessmanagementsysteme an verwandte Systeme beispielsweise zur Ressourcenverwaltung oder Middlewaresystemen, beispielsweise so genannten Enterprise Service Busses (ESB), leichter, da hierfür bereits bestimmte Funktionen oder Programmierschnittstellen vorhanden sind.

Standardtreue Kommerzielle Prozessmanagementsysteme wie der WebSphere Process Server weisen eine Standardkonformität auf, die von reinen Forschungsprototypen normalerweise nicht erreicht wird. Der WebSphere Process Server beispielsweise verwendet für Prozessdefinitionsmodelle den Standard WS-BPEL und ermöglicht so die Austauschbarkeit dieser Modelle mit anderen Prozessmanagementsystemen, beispielsweise dem Oracle BPEL Process Manager [RW09, Kapitel 5].

Höhere Zuverlässigkeit Forschungsprototypen erreichen normalerweise nicht die Reife oder Fehlertoleranz, die kommerzielle Systeme aufweisen. Der Grund hierfür liegt wiederum darin, dass die Sicherstellung auch dieser Qualitätsmerkmale mit hohen Aufwänden aber geringem wissenschaftlichen Profit einhergeht.

Höhere Akzeptanz Es ist günstig, auf ein bestehendes, bereits im Einsatz befindliches System aufzusetzen, da die betroffenen Anwender hier nicht umlernen müssen bzw. die Entwickler und Administratoren kein grundsätzlich neues Prozessmanagementsystem in die übrige Systeminfrastruktur einbetten müssen. Somit kann von beiden Gruppen eine höhere Akzeptanz erwartet werden im Vergleich zur Einführung einer Neuentwicklung.

Die Liste der Vorteile ließe sich auf plausible Weise noch um weitere Punkte ergänzen. Zusammenfassend und in Hinblick auf die bekannten Qualitätsmerkmale von Software [Bal98, Seite 261] kann gesagt werden, dass die Qualität bei Verwendung bestehender Systeme allgemein höher ist. Hierbei kann unter “Verwendung” zweierlei verstanden werden:

Änderung bestehender Prozessmanagementsysteme Ein bestehendes, statisches Prozessmanagementsystem derart zu ändern, dass auch dynamische Prozesse angemessen unterstützt werden, ist ein mögliches Vorgehen. In diesem Fall muss die *Implementierung* dahingehend geändert werden, dass Abweichungen einer Prozessinstanz vom Prozessdefinitionsmodell durch entsprechende, zusätzliche Funktionsaufrufe unterstützt werden. Voraussetzung ist in diesem Fall selbstverständlich, dass das verwendete bestehende System quelloffen ist. Wird ein bestehendes Prozessmanagementsystem geändert statt nur erweitert, kann die *Qualität der Gesamtarchitektur* beibehalten werden.

Erweiterung bestehender Prozessmanagementsysteme Ist der Quellcode des verwendeten bestehenden Prozessmanagementsystems wie in dieser Arbeit nicht zugreifbar, so bleibt nur die Möglichkeit, das bestehende System mittels der *vorhandenen Programmierschnittstellen* zu erweitern. Damit bleibt das bestehende System in sich unverändert. Die Erweiterung ist somit auch zwangsweise optional, d.h. die *Funktionalität* des erweiterten Systems bleibt *unverändert erhalten*.

Neben den genannten Vorteilen gibt es weitere Gründe, weswegen in dieser Arbeit genau dieser Ansatz verfolgt wird. Die Arbeit ist in eine Forschungskooperation mit einem Industriepartner, der Generali Deutschland Informatik Services GmbH eingebettet. Da die Generali Deutschland Informatik Services GmbH das Prozessmanagementsystem WebSphere Process Server schon des Längeren zur Unterstützung von Versicherungsprozessen einsetzt, ist eine strikte Rahmenbedingung der Kooperation die Weiterverwendung dieses Systems. Die Entscheidung, ein bestehendes System zu erweitern statt einen Forschungsprotoypen grundsätzlich mit herkömmlichen oder fortschrittlichen und experimentellen Mitteln neu zu entwickeln, gründet sich hauptsächlich auf dieser Rahmenbedingung des Kooperationsprojekts.

3.7.3 Mögliche Erweiterungen

Unterstützung weiterer Änderungsmuster

Die realisierte Unterstützung der Änderungsmuster lässt sich um weitere ergänzen:

paralleles Einfügen Beim dynamischen Einfügen von Aktivitäten ist immer eine Bezugsaktivität anzugeben, relativ zu der in der Kontrollflussdefinition die neue Aktivität eingefügt wird. Die Erweiterung der Prozessdefinitionsmodelle ist dabei so gehalten, dass die neue Aktivität immer direkt vor bzw. direkt nach der Bezugsaktivität eingefügt werden kann, da je eine

DAI-Aktivitäten sequentiell vor und nach jeder Aktivität der Ursprungsprozessdefinition eingefügt wird. Denkbar wäre, den Teil der Transformation so zu erweitern, dass auch paralleles dynamisches Einfügen möglich ist. Auf einfache Weise lässt sich dies allerdings nur realisieren, wenn die Parallelität der eingefügten Aktivität A auf eine Bezugsaktivität B beschränkt bleibt. Das bedeutet, dass A nur parallel zu B ausgeführt werden kann, Vorgänger- und Nachfolgeraktivitäten von B aber auch Vorgänger- bzw. Nachfolgeraktivitäten von A sind.

temporäre Änderungen Änderungen, die durch dynamisches Einfügen und Löschen zustande gekommen sind, sind insoweit permanent, als dass sie auch bei Wiederholung der geänderten Ablaufstruktur gültig sind. Dabei ist es unerheblich, ob die Wiederholung durch eine bereits im ursprünglichen Prozessdefinitionsmodell definierten Schleife zustande kommt, oder durch einen dynamischen Rücksprung. Eine naheliegende Erweiterung wäre, auch Änderungsoperationen zu realisieren, über die Aktivitäten temporär eingefügt bzw. gelöscht werden können. Dynamische Rücksprünge sind im Gegensatz zum dynamischen Einfügen und Löschen immer temporär. Ein anderes Verhalten ist für diese Änderungsoperation nicht sinnvoll. Die Erweiterung des Ansatzes um temporäre dynamische Einfüge- und Löschoperationen bedarf allein der Änderung der Dynamik-Komponente. In dieser ist bei einer temporär dynamisch eingefügten Aktivität beispielsweise dafür zu sorgen, dass die dynamische Bindung nach einmaliger Ausführung des aufgerufenen Prozessdefinitionsmodells wieder aufgelöst wird.

Datenflussbehandlung

Ein Problem beim dynamischen Einfügen sind die Datenflüsse, an denen die dynamische eingefügte Aktivität oder eine ihrer Kindaktivitäten beteiligt sind. Hier ist es in der derzeitigen Realisierung noch nötig, händisch Prozessvariablen den Formalparametern der eingefügten Aktivität zuzuordnen. Verbessern ließe sich dies durch eine Heuristik, die eine vorläufige Zuordnung vornimmt. Die Heuristik kann dabei Typkompatibilitäten und Namensgleichheiten bzw. -ähnlichkeiten von Prozessvariablen in Betracht ziehen. Eine Vollautomatisierung der Zuordnung lässt sich hingegen nicht erreichen.

3.8 Verwandte Arbeiten

Viele verwandte Forschungsarbeiten beschäftigten sich mit der geeigneten Unterstützung dynamischer Geschäftsprozesse durch neuartige Prozessmanagementsysteme. Trotz der ähnlichen Zielsetzung unterscheiden sich die Ansätze teils fundamental untereinander und von unserem. In diesem Abschnitt werden existierende Ansätze kategorisiert, beschrieben und mit unserem verglichen.

3.8.1 Modellierungszeit-Flexibilität

Im Folgenden werden Ansätze beschrieben, die der Dynamik in Geschäftsprozessen dadurch begegnen, indem sie Prozessmodellierungssprachen einsetzen, deren Modelle keine oder nur an einigen Stellen Vorgaben bzgl. der Ausführungsreihenfolge der modellierten Aktivitäten treffen.

Es kann vorweggenommen werden, dass Sprachen mit extrem hoher Modellierungszeit-Flexibilität für den Projektkontext dieser Arbeit ungeeignet sind. Mit ihrer erhöhten Flexibilität geht generell auch ein höherer Bedarf menschlicher Interaktion einher, die Nichtdeterminismen bei der Prozessdurchführung auflösen muss. Damit sind diese Sprachen für teilweise automatisierte Prozesse unbrauchbar.

Deklarative Sprachen

Prozessdefinitionsmodelle in imperativen Sprachen wie WS-BPEL modellieren insbesondere den Verhaltensaspekt von Prozessen. Dies geschieht in diesen Sprachen derart, dass mögliche Abläufe aus den Prozessdefinitionsmodellen explizit hervorgehen. Prozessdefinitionsmodelle in deklarativen Sprachen hingegen beinhalten lediglich Einschränkungen an ansonsten beliebige Abläufe. Mögliche Abläufe gehen aus diesen Prozessdefinitionsmodellen daher nur indirekt hervor. Ein tiefergehender Vergleich zwischen deklarativen und imperativen Prozessmodellierungssprachen findet sich in [SMR$^+$08].

Ein repräsentativer Vertreter im Bereich deklarativer Prozessmodellierungssprachen ist der DECLARE-Ansatz [PSA07]. DECLARE ist ein Werkzeug zum Entwurf von deklarativen Prozessmodellierungssprachen und zugehöriger Ausführungsunterstützung. Modelle in DECLARE-Sprachen ähneln stark den Prozesswissensmodellen aus Unterabschnitt 2.2.3, können jedoch direkt ausgeführt werden. Eine genauere Beschreibung von DECLARE findet sich in Abschnitt 4.7.

Hybride Sprachen

Komplexe Aktivitäten mit dynamischer Realisierung (Sadiq et al.) Sadiq et al. [SSO01] gehen von der Grundannahme aus, dass ein Prozess auf grobgranularer Ebene immer vorhersagbar abläuft und nur auf feingranularer Ebene unvorhersagbar.

Ansatz In der von Sadiq et al. vorgeschlagenen Prozessmodellierungssprache ist die Realisierung einiger komplexer Aktivitäten – in der Terminologie der Autoren “pockets of flexibility” – auf grobgranularer Ebene zur Modellierungszeit offengelassen. Solchen Aktivitäten mit unbestimmter Realisierung können zur Modellierungszeit Mengen von Aktivitäten zugeordnet werden, die zur Laufzeit jedoch noch beliebig geändert werden können genauso wie die Kontrollflussdefinition innerhalb der komplexen Aktivitäten.

Übereinstimmungen Die “pockets of flexibility” ähneln in gewisser Weise den DAI-Aktivitäten in unserem Ansatz, da diese durch weitere Prozessdefinitionsmodelle zur Laufzeit dynamisch verfeinert werden können. Allerdings sind DAI-Aktivitäten sowohl zur Modellierungszeit als auch zur Laufzeit vom Modellierer bzw. Prozessbeteiligten verborgen und somit müssen auch keine Stellen im Vorhinein identifiziert werden, an denen ein Prozess ggf. dynamisch abläuft.

Unterschiede Im Unterschied zu Sadiq et al. fußt unser Ansatz nicht auf der Annahme, dass man zur Modellierungszeit bereits statische Teile eines Prozesses strikt von dynamischen Teilen trennen kann. Insbesondere gilt in unserem Ansatz nicht die Einschränkung, dass die dynamisch verfeinerbaren Stellen in der Kontrollflussdefinition des grobgranularen, statischen Prozesses zur Modellierungszeit identifiziert werden können. Des Weiteren sind “pockets of flexibility” insofern einstufig, als dass die darin dynamisch eingefügten Aktivitäten atomar sind und keine komplexen Unterprozesse, die wiederum “pockets of flexibility” enthalten können.

Worklets Adams et al. bereichern mit Worklets die Menge verfügbarer hybrider Prozessmodellierungssprachen [Ada07, AHEA06, AHAE07]. Vergleichbar mit den “pockets of flexibility” von Sadiq et al. wird ein Prozessdefinitionsmodell an zur Modellierungszeit festgelegten Stellen um andersartige Konstrukte erweitert.

Ansatz Diese andersartigen Konstrukte sind in diesem Fall Worklets, die binäre Bäume darstellen. Jeder Baumknoten k besitzt neben einem Verweis auf ein Prozessdefinitionsmodell p eine Bedingung c. Wird ein Worklet ausgeführt, so werden beginnend mit der Baumwurzel die Bedingungen ausgewertet und ggf. die Prozessdefinitionsmodelle ausgeführt. Ist für k die Bedingung c erfüllt, so wird mit dem rechten Kind von k fortgefahren, sofern existent. Ist c nicht erfüllt wird mit dem linken Kind von k fortgefahren, wenn es dieses gibt. Ist ein Blatt erreicht und dessen Bedingung erfüllt, wird das darin referenzierte Prozessdefinitionsmodell durchgeführt. Worklets können als Verfeinerung von Aktivitäten imperativer YAWL-Prozessdefinitionsmodelle [AH05] verwendet werden. Die Bedingungen in den Worklets nehmen Bezug auf Prozessvariablen in den YAWL-Prozessdefinitionsmodellen. Dynamisch änderbar sind bei diesem Ansatz nur die Worklets, d.h. die Baumstruktur kann um weitere Knoten ergänzt werden.

Unterschiede Worklets bieten nur sehr eingeschränkte Mittel zur Behandlung von Dynamik in Prozessen. Wiederum müssen zur Modellierungszeit die Aktivitäten identifiziert werden, die durch änderbare Worklets realisiert werden. Der Nutzen von Worklets ist daher eher in solchen Prozessen zu sehen, in denen die Dynamik hauptsächlich in den Bedingungen zu finden ist, unter denen eine Aktivität ausgeführt wird. Hier können über das zusätzliche Sprachmittel der Worklets komplizierte Verzweigungsstrukturen in YAWL-Prozessdefinitionsmodellen vermieden werden, die zudem noch zur Laufzeit dynamisch anpassbar sein müssen.

3.8.2 Laufzeit-Dynamik

Im Folgenden werden einige Ansätze beschrieben, die ähnlich zu unserem Ansatz Dynamik in Geschäftsprozessen durch strukturelle Änderbarkeit der Prozessinstanzmodelle behandeln.

Kommerzielle Systeme Kommerzielle Prozessmanagementsysteme unterstützen dynamische Geschäftsprozesse gar nicht oder nur unzureichend.

Staffware Das verbreitete Prozessmanagementsystem Staffware unterstützt dynamische Änderungen gar nicht [Rei00, Seite 11]. Es ist zwar rein technisch möglich, die Prozessdefinitionsmodelle laufender Instanzen auszutauschen, die Effekte auf laufende Instanzen sind dabei jedoch nicht vorhersagbar und betreffen immer alle Instanzen, was im Allgemeinen ungewollt ist. Aus dem gleichen Grund wurde ein vergleichbares Vorgehen in dieser Arbeit auf Basis des WebSphere Process Servers nicht gewählt.

InConcert InConcert ist ein vergleichbar verbreitetes Prozessmanagementsystem. Im Unterschied zu Staffware werden einfache dynamische Änderungen in laufenden Instanzen direkt unterstützt. Dies wird allerdings dadurch erkauft, dass die Ausdrucksmächtigkeit der Prozessmodellierungssprache von InConcert gegenüber WS-BPEL sehr begrenzt ist. Schleifen sind beispielsweise nicht modellierbar [Rei00, Unterabschnitt 2.3.2.2].

Beiden Systemen ist gemein, dass dynamische Änderungen ohne jedwede Unterstützung für den Prozessbeteiligten durchgeführt werden müssen. Technische und fachliche Probleme müssen von diesem selbst erkannt und vermieden werden. Außerdem verwenden beide Systeme proprietäre Prozessmodellierungssprachen statt auf einem Sprachstandard wie WS-BPEL aufzusetzen.

AHEAD Das AHEAD-System wurde bereits in Abschnitt 2.7 beschrieben. Trotz der Tatsache, dass AHEAD eine Vorgängerarbeit zu dieser Arbeit darstellt, sind die Unterschiede zu dieser Arbeit hinsichtlich der Umsetzung von Laufzeit-Dynamik zahlreich. AHEAD wurde von Grund auf neu entwickelt und ist insofern ein typischer Vertreter der in Unterabschnitt 3.7.1 beschriebenen Klasse von Prozessmanagementsystemen mit den erwähnten Vor- und Nachteilen.

Prozessfragmentkomposition zur Laufzeit (Mangan et al.) In [MS03] erläutern Mangan et al. einen Ansatz zur Unterstützung von dynamischen Prozessen durch kontinuierliche Evolution eines Prozessinstanzmodells zur Laufzeit durch Fragmente von Prozessdefinitionsmodellen.

Ansatz Der Ansatz basiert auf einer einfachen Prozessmodellierungssprache, die neben Ausführungspräzedenzen nur die Modellierung von Verzweigungen und Parallelitäten vorsieht. Fragmente in dieser Prozessmodellierungssprache können zur Prozesslaufzeit miteinander kombiniert werden, so dass typischerweise in einem laufenden Prozess gewissermaßen nur die nahe Zukunft modelliert ist. Die Kombinierbarkeit von Fragmenten kann durch separat modellierte Bedingungen eingeschränkt werden. Eine solche Bedingung kann beispielsweise erzwingen, dass eine Aktivität E erst dann ausgeführt werden darf, wenn Aktivitäten A und B ausgeführt wurden.

Gemeinsamkeiten Die von Mangan et al. verfolgte ad-hoc-Modellierung von Prozessen zur Prozesslaufzeit ist mit unserem Ansatz prinzipiell auch möglich, wenn auch eine eher untypische Form der Verwendung. Die Erzwingung der Einhaltung bestimmter Bedingungen wird von unserem Ansatz ebenfalls unterstützt (vgl. Kapitel 4).

Unterschiede Die Arbeit von Mangan et al. basiert auf einer eigens entwickelten Prozessmodellierungssprache. Auf bestehende Sprachen wurde keine Rücksicht genommen. Es lassen sich zudem im Ansatz von Mangan et al. nur sehr einfache Bedingungen ausdrücken. Eine Evaluierung des Ansatzes in Form einer prototypischen Implementierung wurde bis jetzt nicht veröffentlicht.

ADEPT und Aristaflow Unter den Ansätzen, die Änderungen an Prozessinstanzmodellen zur Laufzeit zur Behandlung von Dynamik vorsehen, ist der ADEPT-Ansatz [RD98, Rei00] von Reichert et al. neben dem AHEAD-Ansatz der elaborierteste. Zentral im ADEPT-Ansatz ist eine eigens entwickelte Prozessmodellierungssprache, in der Prozessinstanzmodelle und somit, wie in unserem Ansatz, auch Prozessdefinitionsmodelle ausgedrückt werden können. Die Konzepte wurden in das kommerzielle Prozessmanagementsystem Aristaflow [AD07, DR09] überführt.

Ansatz Das formale Fundament des ADEPT-Ansatzes ist ein Metamodell für Prozessinstanzmodelle ("Kontrollflussgraphen" in der Terminologie der Autoren), vergleichbar mit dem aus Unterabschnitt 2.3.1. Ein formales Graphkalkül beschreibt mögliche Änderungen an Prozessinstanzmodellen, wobei die Änderungen einerseits nur Prozessfortschritte durch Veränderung von Aktivitätszuständen sein können, andererseits aber auch strukturelle Änderungen an der Kontrollflussdefinition im Prozessinstanzmodell zur Behandlung von Dynamik im laufenden Prozess. Das Graphkalkül ist so spezifiziert, dass strukturelle Änderungen in korrekten Prozessinstanzmodellen stets wieder zu korrekten Prozessinstanzmodellen führen. Infolgedessen wird ein Prozessbeteiligter bei strukturellen Änderungen im Prozessinstanzmodell syntaktisch vom ADEPT-Prozessmodelleditor geführt, so dass syntaktisch unkorrekte Prozessinstanzmodelle gar nicht erst konstruiert werden können.

Gemeinsamkeiten Der ADEPT-Ansatz ähnelt unserem und Vorarbeiten im AHEAD-Projekt unter allen verwandten Arbeiten am stärksten. Die Menge der für Prozessinstanzmodelle verwendbaren Kontrollflusskonstrukte und abgedeckte Aspekte wie Datenflüsse ist mit denen von WS-BPEL und folglich mit denen von SimBPEL vergleichbar. Übereinstimmend mit SimBPEL-Instance können in Kontrollflussgraphen auch Prozesszustandsinformationen modelliert werden. Prozessmodellierungssprachen anderer Ansätze beschränken sich auf die Modellierung von Prozessdefinitionsmodellen. Die Unterstützung von Prozessbeteiligten besteht darin, nur syntaktisch korrekte dynamische Änderungen in Kontrollflussgraphen durchzuführen. Dieses wird von unserem Ansatz auch berücksichtigt, wie in Abschnitt 4.3 erläutert.

Unterschiede Der wesentliche Unterschied unseres Ansatzes zum ADEPT- und Aristaflow-Ansatz (und allen weiteren hier beschriebenen verwandten

Arbeiten) ist die Tatsache, dass unser Ansatz auf einem bestehenden System und einem Sprachstandard aufbaut mit den bereits diskutierten vielen Vor- und wenigen Nachteilen. Die Einhaltung der syntaktischen Korrektheit wird von unserem Ansatz zwar auch unterstützt, ist aber auf andere Weise realisiert (vgl. Abschnitt 4.3) und geht über rein technische Bedingungen an Prozesse hinaus (vgl. Abschnitt 4.4).

Geplante dynamische Änderungen (Weske et al.) Der WASA-Ansatz von Weske et al. [Wes99a, VW98] basiert ebenfalls auf einer eigens entwickelten, einfachen Modellierungssprache, die stark vereinfachten Aufgabennetzen des AHEAD-Ansatzes ähnelt.

Ansatz Der WASA-Ansatz ähnelt dem von ADEPT. Wiederum steht eine proprietäre Prozessmodellierungssprache zur Darstellung von Prozessmodellinstanzen im Vordergrund. Die WASA-Modellierungssprache ist ähnlich wie die Sprachen des AHEAD-Ansatzes ganzheitlich in dem Sinne, dass die verschiedenen Modellierungsaspekte nach Jablonski et al. [JB96] abgedeckt sind. Als Besonderheit lassen sich spezielle Planungsaufgaben in den Prozessmodellen modellieren. Diese Planungsaufgaben modellieren keine fachliche Aktivität des Prozesses, sondern markieren die Stellen im Prozess, an denen das Prozessinstanzmodell zur Prozesslaufzeit erweitert werden muss. Diese Idee ist vergleichbar mit dem Zustand InDefinition von Aktivitäten in Aufgabennetzen im AHEAD-Ansatz (vgl. Abschnitt 2.7). Eine Planungsaufgabe impliziert also bereits zur Modellierungszeit, dass eine dynamische Änderung zur Laufzeit durchgeführt werden muss. Planungsaufgaben sind in WASA allerdings nicht notwendig für dynamische Änderungen; diese lassen sich auch so zur Prozesslaufzeit durchführen.

Unterschiede Die entwickelte Prozessmodellierungssprache fällt durch ihre Sparsamkeit bzgl. der verwendbaren Kontrollflusskonstrukte auf. Im Wesentlichen lassen sich unter diesem Aspekt nur Ausführungspräzedenzen modellieren, ähnlich zu den Aktivitäten in SimBPEL-Instance-Modellen innerhalb von `Flow`-Elementen. Planungsaufgaben sind in unserem Ansatz nicht vorhanden, was auf die Ansicht zurückzuführen ist, dass bei dynamischen Prozessen insbesondere der Zeitpunkt, an dem eine dynamische Änderungen durchzuführen ist, nicht sinnvoll eingegrenzt werden kann.

WIDE WIDE von Casati et al. [Cas98, CCPP99, CCPP00] ist ein weiterer Vertreter von Prozessmanagementsystemen, die zur Laufzeit dynamische Änderungen in Prozessinstanzmodellen erlauben. Auch hier wird wiederum eine nicht-standardisierte, selbst entwickelte Modellierungssprache eingesetzt [CCPP00].

Gemeinsamkeiten Am WIDE-Ansatz im Vergleich zu unserem Ansatz fällt auf, dass sich bestimmte dynamische Änderungen in den WIDE-Prozessinstanzmodellen teils so wiederfinden, wie sie durch die Dynamik-Schicht verborgen vom Prozessbeteiligten umgesetzt werden. Die dynamische Löschung einer Aktivitäten B aus einer Sequenz von Aktivitäten A, B, C wird in WIDE beispielsweise so umgesetzt, dass zwischen A und C eine Verzweigung eingefügt wird und B darin in einen – aufgrund der Verzweigungsbedingung – nicht durchführbaren Zweig umgebettet wird. Vergleichbar sorgt die Dynamik-Schicht dafür, dass Aktivitäten innerhalb von DRD-Aktivitäten wahlweise zur Prozesslaufzeit durchgeführt oder umgangen werden können.

Unterschiede Die beschriebene Umsetzung von beispielsweise dynamischen Löschungen ist für Prozessbeteiligten verwirrend. In unserem Ansatz werden die dynamikerzeugenden Strukturen und insbesondere DRD-Aktivitäten vom Prozessbeteiligten verborgen. Weshalb dieses in WIDE nicht der Fall ist, bleibt unklar, zumal WIDE nicht auf einem bestehenden statischen System wie dem IBM WebSphere Process Server aufsetzt.

Späte Modellierung und Binden von Unterprozessen (Han et al. und Hagemeyer et al.) Zwei untereinander sehr ähnliche Arbeiten sind der HOON-Ansatz [Han97] und der MOVE-Ansatz [HJH96, HHJS97]. Beide erlauben es, Prozessdefinitionsmodelle zu dekomponieren. Eine Prozessdurchführung bewirkt somit eine Kette von Instanziierungen von Prozessdefinitionsmodellen und Unterprozessdefinitionsmodellen. Dynamik kann in beiden Ansätzen so behandelt werden, dass Unterprozessdefinitionen vor ihrer Instanziierung noch geändert werden bzw. Bindungen zwischen aufrufendem Prozessdefinitionsmodell und aufgerufenen Unterprozessdefinitionsmodellen ausgetauscht werden können. Der MOVE-Ansatz erlaubt darüber hinaus, Unterprozessinstanzmodelle zu deren Laufzeit dynamisch zu ändern. Hierfür müssen allerdings zuvor zur Modellierungszeit diejenigen Aktivitäten im aufrufenden Prozessdefinitionsmodell identifiziert werden, für deren aufgerufene Unterprozessinstanzmodelle dies möglich sein soll.

Abgrenzung Spätes Binden oder die Begrenzung von dynamischen Änderungen auf bestimmte Stellen im Prozess, d.h. bestimmte Unterprozesse, reicht für viele Situationen nicht aus. Solche Einschränkung implizieren, dass zumindest die Position von dynamischen Änderungen im Prozess vorhergesehen werden kann, was aber häufig nicht der Fall ist.

3.8.3 Konfigurierung von Maximalmodellen

Im erweiterten Sinne verwandt mit unserem Ansatz sind Ansätze, die sich mit der Konfigurierung von Prozessdefinitionsmodellen auseinandersetzen [RDH08, SP06, GAJVR08, HBR08]. Sie gehen von der durch unseren Ansatz nicht vertretenen Hypothese aus, dass sich Geschäftsprozesse a-priori zur Modellierungszeit vollständig in Maximalmodellen spezifizieren lassen (vgl. Unterabschnitt 1.2.2). Der dadurch entstehenden Kompliziertheit der Prozessdefinitionsmodelle begegnen sie durch einen Konfigurierungsschritt, der chronologisch zwischen bisheriger Modellierungs- und -laufzeit eines Prozesses einzuordnen ist.

La Rosa et al. beschreiben in [RDH08] einen typischen Vertreter dieser Ansätze. Ihr Vorgehen ist, während des Konfigurierungsschritts spezielle Verzweigungen bereits zu evaluieren und nicht-aktivierte Zweige aus dem maximalen Prozessdefinitionsmodell zu entfernen. Ähnlich gehen Rosemann et al. in [RA07] vor. Die Sprache der Ereignisgesteuerten Prozessketten (EPCs) für Prozessdefinitionsmodelle wird erweitert zu “Konfigurierbare Ereignisgesteuerte Prozessketten”. In diesen lassen sich besagte Maximalmodelle spezifizieren, deren Aktivitäten so annotiert werden können, dass sie je nach Parametrisierung zur Konfigurierungszeit aus dem Prozessdefinitionsmodell ausgeblendet werden bzw. eingeblendet bleiben. Ist eine Aktivität ausgeblendet, bleibt sie zu Laufzeit effektfrei. Gottschalk et al. [GAJVR08] verlegen die Konfigurierung auf die Kontrollflussdefinitionen eines Maximalmodells statt (komplexe) Aktivitäten ein- oder auszublenden. Kontrollflüsse können zur Konfigurierungszeit unpassierbar gemacht werden; unerreichbare Aktivitäten werden infolgedessen entfernt. Ferner ist es möglich, bzgl. Kontrollflussdefinition inzidente Aktivitäten zu verschmelzen.

Konfigurierungsansätze sind nicht brauchbar zur angemessenen Unterstützung dynamischer Geschäftsprozesse. Vielmehr reduzieren sie die in einem Maximalmodell bereits enthaltene Flexibilität und wirken besagter Unterstützung sogar entgegen. Sie können jedoch sinnvoll mit Ansätzen wie unserem kombiniert werden in der Art, dass ein Maximalmodell zur Konfigurierungszeit zwecks Übersichtlichkeit für den Prozessbeteiligten auf erwartbare Ausführungspfade reduziert wird, eine Abweichung durch dynamische Änderungen zur Laufzeit jedoch möglich bleibt.

3.8.4 Kopplung von Prozessmanagementsystemen

Diverse Vorarbeiten im AHEAD-Projekt und parallele Arbeiten im PROCEED-Projekt sind mit dieser Arbeit insoweit verwandt, als dass sie eine Kopplung zwischen Editoren Dynamischer Aufgabennetze (vgl. Abschnitt 2.7) und Prozessinstanzen anstreben, die in separaten, teils kommerziellen Systemen ausgeführt werden, die dynamische Änderungen nicht oder nur auf Umwegen unterstützen. Diese Arbeiten unterscheiden sich von dieser Arbeit jeweils hauptsächlich darin, dass in dieser Arbeit nicht mehrere eigenständige Systeme miteinander gekoppelt werden. Stattdessen wird der bestehende WebSphere Process Server lediglich um eine Schicht erweitert, die, unter Beibehaltung der Sprache WS-BPEL, die Möglichkeit zu dynamischen Änderungen in Prozessinstanzmodellen simuliert. Daher muss das Problem der Integration von Prozessinstanzmodellen verschiedener Sprachen, beispielsweise Dynamischer Aufgabennetze und Petri-Netze, nur in diesen verwandten Arbeiten gelöst werden, wohingegen in dieser Arbeit die Integration von statischen Prozessinstanzmodellen (bzw. Prozessdefinitionsmodellen zzgl. Prozessinstanzdaten) und dynamischen Prozessinstanzmodellen zu lösen war.

Dynamische Aufgabennetze und COSA-Prozessdefinitionsmodelle

Becker und Jäger beschreiben in [Bec01, Jäg03] die Kopplung des AHEAD-Aufgabennetzeditors für Dynamische Aufgabennetze mit dem kommerziellen Prozessmanagementsystem COSA [COS]. Die Kopplung bezweckt, dass der AHEAD-Aufgabennetzeditor nur noch für Darstellungs- und Editierungszwecke verwendet wird, die Prozessinstanzmodellinformationen hingegen in COSA vorgehalten werden. Im Groben folgt die Kopplung folgenden Schritten:

1. Ein initiales Dynamisches Aufgabennetz A wird im AHEAD-Aufgabennetzeditor modelliert, wobei sich alle Aufgaben noch in einem Vorbereitungszustand befinden (vgl. Abbildung 2.30).

2. Das Dynamische Aufgabennetz A wird in ein COSA-Prozessdefinitionsmodell (Petri-Netz) C transformiert.

3. Das COSA-Prozessdefinitionsmodell C wird instanziiert, d.h. ein COSA-Prozessinstanzmodell C_i erzeugt.

4. Änderungen der Ausführungszustände im COSA-Prozessinstanzmodell C_i werden fortwährend zum AHEAD-Aufgabennetzeditor propagiert und im Dynamischen Aufgabennetz A dargestellt.

5. Im Falle einer dynamischen Änderung wird zunächst die Ausführung in COSA gestoppt.

6. Dynamische Änderungen werden im AHEAD-Aufgabennetzeditor durchgeführt.

7. Das geänderte Dynamische Aufgabennetz A' wird erneut in ein COSA-Prozessdefinitionsmodell C' transformiert.

8. Das COSA-Prozessdefinitionsmodell C' wird zu C'_i instanziiert, wobei die Tokenmarkierung in C'_i direkt so angepasst wird, dass sie dem bereits fortgeschritten Ausführungszustand des Prozesses entspricht.

Gerade der letzte Schritt offenbart den Hauptunterschied zum Verfahren dieser Arbeit. Im WebSphere Process Server ist es nicht möglich, Prozessdefinitionen laufender Instanzen auszutauschen und Ausführungsinformationen laufender Instanzen direkt zu ändern. Daher wurde in dieser Arbeit ein Ansatz gewählt, der diese Art von Änderungsmöglichkeit nicht voraussetzt.

Dynamische Aufgabennetze und XPDL-Prozessdefinitionsmodelle

In [Hel08] beschreibt Heller die Kopplung des AHEAD-Aufgabennetzeditors mit dem Prozessmanagementsystem Shark [sha]. Im Unterschied zur Kopplung mit COSA, verbleibt ein Teil des Gesamtausführungszustands eines Prozesses innerhalb des AHEAD-Aufgabennetzeditors. Lediglich Verfeinerungen bestehender Aufgaben können durch ein in Shark vorgehaltenes XPDL-Prozessdefinitionsmodell modelliert und ausgeführt werden. Fortschrittsinformationen von Prozessinstanzen innerhalb von Shark werden im Dynamischen Aufgabennetz von AHEAD widergespiegelt, wobei hierbei die Sprachunterschiede zwischen DYNAMITE und XPDL, wie beispielsweise nur in XPDL vorhandene Schleifen, überbrückt werden müssen. Die Einbeziehung einer weiteren Sprache WOMS für Prozessdefinitionsmodelle verfahrenstechnischer Arbeitsprozesse [SG03] zur Vervollständigung eines ganzheitlichen Modellierungsansatzes, wie in [HHH$^+$08] dargelegt, ist in [HHM$^+$06] erläutert.

Die Arbeit von Heller ist durch die Verteilung des Ausführungszustands eines Prozesses auf zwei Systeme und Modelle unterschiedlicher Sprachen ansatzbedingt deutlich von dieser Arbeit verschieden.

Dynamische Aufgabennetze und WF-Prozessdefinitionsmodelle

Einen Schritt weiter als Heller gehen Heer et al. in [HBW08]. Die Kopplung zwischen AHEAD und Shark ist nur einstufig in dem Sinne, dass ein XPDL-Prozessdefinitionsmodell nur eine AHEAD-Aufgabe verfeinern kann, aber nicht umgekehrt. Dynamische Aufgabennetze, die im PROCEED-Aufgabennetzeditor modelliert und ausgeführt werden, können hingegen mit WF-Prozessinstanzen so gekoppelt werden, dass sowohl WF-Prozessdefinitionsmodelle Verfeinerungen von Aufgaben in Dynamischen Aufgabennetzen sein können als auch WF-Aktivitäten wiederum durch Dynamische Aufgabennetze verfeinert werden können. Insbesondere können feingranulare Informationen aus Modellen beider Sprachen für mannigfaltige Darstellungen des Prozessfortschritts verwendet werden [HAW09].

3.8.5 Fazit

In Tabelle 3.1 sind die wichtigsten Vergleichskriterien für dynamische Prozessmanagementsysteme zusammengefasst. Die Tabelle umfasst dabei neben unserem Ansatz nur die ähnlichsten verwandten Arbeiten. Ansätze, die Dynamik durch erhöhte Flexibilität in der Prozessmodellierungssprache beantworten, sind schlecht mit unserem Ansatz vergleichbar und daher nicht enthalten.

Auffällig ist, dass nur wenige Systeme gleichsam eine ausdrucksstarke Prozessmodellierungssprache wie WS-BPEL bereitstellen zugunsten der Flexibilität zur Modellierungszeit und gleichzeitig dynamische Änderungen zur Laufzeit erlauben. In unserem Ansatz verfolgen wir einen Mittelweg, indem eine ausdrucksstarke Sprache WS-BPEL eingesetzt wird und gebräuchliche Änderungsmuster zur Laufzeit bereitgestellt werden. Der AHEAD-Ansatz demgegenüber verzichtet beispielsweise auf Schleifen und Verzweigungen, bietet also nicht die Möglichkeit, vorab alternative und iterative Ausführungen zu planen. Der Grund hierfür liegt darin begründet, dass AHEAD auf hochdynamische Entwicklungsprozesse ausgerichtet ist, in denen typischerweise nur die im aktuellen Prozess tatsächlich auch durchgeführten Aktivitäten sinnvollerweise modelliert werden.

Sämtliche verwandte Forschungsarbeiten ignorieren bestehende Standards und Systeme völlig. Die Vor- und Nachteile wurden am Anfang des Kapitels bereits diskutiert. Einschlägigen Veröffentlichungen ist nicht zu entnehmen, inwieweit die Forschungsprototypen zur Marktreife gebracht wurden und wie groß ggf. die jeweilige Relevanz in Marktbereich der Prozessmanagementsysteme ist. In den meisten Fällen scheint es, als ob ein Industrietransfer nicht angestrebt wurde. Allein die kommerziellen Systeme Aristaflow und FLOWer können als Nachfahren von Forschungsarbeiten angesehen werden. [DR09] ist indirekt zu entnehmen, dass ein unmittelbarer Einsatz von Aristaflow in der industriellen Praxis allerdings noch aussteht im Gegensatz zu FLOWer, das laut [AW05, Abschnitt 6] bereits im Versicherungsumfeld eingesetzt wurde. Bei FLOWer ist jedoch im Gegensatz zu Aristaflow unklar, wie stark es auf zuvor erforschten Konzepten basiert.

In fast keiner Veröffentlichung zu verwandten Ansätzen ist explizit beschrieben, welche Aktivitäten in den Prozessen automatisch oder manuell durchgeführt werden und ob überhaupt beide Realisierungsformen vorgesehen sind. Es kann daher nicht genau beurteilt werden, ob vollautomatische Prozesse (A2A-Prozesse), deren Aktivitäten gänzlich durch Softwareteilsysteme realisiert werden oder manuelle, in den Aktivitäten nur durch Menschen durchgeführt (P2P) werden oder auch Mischformen aus beiden unterstützt werden (P2A). Nur AHEAD stellt schon durch seine Namensgebung (Adaptable and Human-Centered Environment for the MAnagement of Development Processes) klar, dass die Aktivitäten in Aufgabennetzen letztlich von Menschen

	Wörzberger	AHEAD	ADEPT	WASA	WIDE	InConcert
Anwendungs-domäne	Geschäfts-prozesse	Entwicklungs-prozesse	Geschäfts-prozesse	Geschäfts-prozesse	Geschäfts-prozesse	Geschäfts-prozesse
Automati-sierung der Durchführung	vollautom., teilautom., manuell	manuell	manuell (autom. in Aristaflow)	manuell	manuell	vollautom. teilautom. manuell
Prozessmodellierungssprache						
Ausdrucks-mächtigkeit	entspricht WS-BPEL	feingran. Ausführungs-präzedenzen	vergleichbar mit WS-BPEL	nur Ausführungs-präzedenzen	vergleichbar mit WS-BPEL	eingeschränkt im Vergleich zu WS-BPEL
Ausführungs-präzedenzen	Ende-Start (aufgrund Festlegung in WS-BPEL)	Ende-Start, Ende-Ende (beliebig erweiterbar)	Ende-Start	Ende-Start	Ende-Start	Ende-Start
Verzweigungen	ja	nein	ja	nein	ja	ja
Schleifen	ja	nein	ja	nein	ja	nein
Modellaspekte	Kontrollfluss, Datenfluss	Kontrollfluss, Datenfluss, Ressourcen, Dokument-versionen	Kontrollfluss, Datenfluss	Kontrollfluss, Datenfluss, Ressourcen. Aspekte,	Kontrollfluss	Kontrollfluss, Datenfluss
Dynamische Änderungen						
dominantes Änderungs-pardigma	Abweichung	fortwährende Evolution	Abweichung	Abweichung, geplante Evolution	Abweichung, geplante Evolution	Abweichung
Wirkungs-bereich von Änderungen	eine Prozess-instanz	eine Prozess-instanz, Migration anderer Instanzen	eine Prozess-instanz, Migration anderer Instanzen	eine Prozess-instanz	alle Prozess-instanzen einer Definition	eine Prozess-instanz
Änderungs-muster	Einfügen, Löschen, Rücksprung	beliebige Kontrollfluss-änderungen, Verfeinerung, Rückgriffe, Evolution	zahlreiche elementare und komplexe Änderungen eines bestehenden Kontroll-flusses	unbekannt	indirekt Einfügen, Löschen	Einfügen Löschen
Modell-prüfungsarten	Korrektheit, Komplianz, Konsistenz	Korrektheit, Komplianz	Korrektheit	Korrektheit	-	-
Integrationsaspekte						
Standardtreue	WS-BPEL, WebServices (durch WPS)	nein	nein	nein	nein	-
Kopplung / Erweiterung bestehender PMS	Erweiterung von WPS	Kopplung mit PMS Shark, COSA	nein	nein	nein	nein

Tabelle 3.1: Dynamische Prozessmanagementsysteme

durchgeführt werden. WASA legt durch seine teils durch Planungsaufgaben erzwungenen dynamischen Änderungen zur Laufzeit nahe, dass die Unterstützung vollautomatisierter Prozesse nicht vorgesehen ist. Andere Ansätze suggerieren durch die Wahl und Erläuterung der Beispiele in den zugehörigen Veröffentlichungen ebenfalls, dass menschlich durchgeführte Prozesse Gegenstand der jeweiligen Arbeiten sind.

WS-BPEL hingegen und der WebSphere Process Server sind in frühen Fassungen für die Unterstützung vollautomatisierter Prozesse entworfen worden. Die Möglichkeit, menschliche Aktivitätsrealisierungen durch so genannte Human-Tasks direkt in WS-BPEL zu modellieren, war ursprünglich nicht vorgesehen. Dies erklärt einerseits die Modellierungszeit-Flexibilität der Sprache WS-BPEL, andererseits allerdings auch die nicht vorhanden Unterstützung dynamischer Änderungen zur Prozesslaufzeit durch den WebSphere Process Server. Aufgrund dieser Ausrichtung sind reichhaltige Integrationsmöglichkeiten zwischen WS-BPEL-Prozessen und Softwareteilsystemen in verschiedensten Formen als Aktivitätsrealisierungen vorhanden und werden durch unseren Ansatz erhalten. Keiner der verwandten Ansätze gleicht dies durch eigene Konzepte in diesem Bereich aus. Dies mag der Hauptgrund sein, weshalb die in diesem Abschnitt beschriebenen, dynamischen Prozessmanagementsystem-Prototypen bislang auf wenig Resonanz in der Industrie gestoßen sind.

Letztlich ist auch das in unserem Ansatz vorherrschende Paradigma dynamischer Änderungen durch die Ursprünge von WS-BPEL zurückzuführen. Generell werden WS-BPEL-Prozesse nach wie vor so statisch zur Modellierungszeit spezifiziert, dass sie auch ohne dynamische Änderungen in sinnvoller Granularität und Durchgängigkeit einen Geschäftsprozess unterstützen können. Dynamik, also Unvorhergesehenes, wird durch dynamische Änderungen der Prozessinstanzmodelle beantwortet. Prinzipiell sind durch fortwährende dynamische Einfügungen auch Unterstützungen wie in AHEAD oder WIDE denkbar, wobei aufgrund der noch fehlenden Möglichkeit zum parallelen Einfügen dann eine sequentielle Prozessdurchführung erzwungen wird. Diese Form der Unterstützung ist jedoch eher atypisch, da die extremen Formen von Unbestimmtheit in den betrachteten Geschäfts- und insbesondere Versicherungsprozessen nicht auftreten.

Kapitel 4

Explizites Prozesswissen

In Kapitel 2 wurden Prozessmodelle für die Modellierung von Prozessen auf verschiedenen Abstraktionsebenen eingeführt. Die detailliertesten hiervon sind Prozessinstanzmodelle, die jeweils einen konkreten Prozess modellieren. Sie werden zur Kommunikation zwischen Prozessbeteiligten und Prozessmanagementssystem verwendet, wie im vorherigen Kapitel dargelegt. Über Prozessinstanzmodelle können dynamische Prozessänderungen von Prozessbeteiligten an das Prozessmanagementssystem kommuniziert werden. Über die Dynamik-Schicht oberhalb des WebSphere Process Servers werden die Änderungen im Prozessinstanzmodell auf die laufenden Instanzen im WebSphere Process Server abgebildet (vgl. Kapitel 3). Die Dynamik-Schicht dient allein dieser technischen Abbildung zwischen Änderungen im lokalen Prozessinstanzmodell des Prozessmodelleditors und den Datenstrukturen im WebSphere Process Server. Gänzlich vernachlässigt wird dabei die Tatsache, dass nicht jede technisch mögliche dynamische Änderung auch sinnvoll ist.

Als sinnvoll werden solche Änderungen in Prozessinstanzmodellen erachtet, die in der weiteren Prozessdurchführung nicht zu technischen oder fachlichen Problemen führen. In Unterabschnitt 1.3.3 wurden einige Beispiele für Änderungen an einem Prozessinstanzmodell aufgeführt, die aus technischer oder fachlicher Sicht problematisch sind. Prozessbeteiligte müssen darin unterstützt werden, nicht-sinnvolle Situationen in Prozessinstanz-, -definitions-, und -wissensmodellen zu erkennen. In diesem Kapitel wird die Konzeption eines Prüfwerkzeugs als Teil des Prozessmodelleditors beschrieben, das dieses leistet: Nicht-sinnvolle Situationen in Prozessmodellen aller Abstraktionsebenen werden erkannt, klassifiziert und mit Verweis auf die jeweils verursachende Stelle im Prozessmodell dem Prozessbeteiligten präsentiert. Insbesondere können so nicht-sinnvolle Situationen in dynamisch geänderten Prozessinstanzmodellen aufgedeckt werden, bevor die Änderungen an die Dynamik-Komponente propagiert werden (vgl. Befehl Commit Changes in Abbildung 3.25).

Die Abbildung 4.1 fasst die in diesem Kapitel erläuterten Zusammenhänge zusammen. Die automatischen Prüfungen sind syntaktische Prüfungen der Prozessmodelle. In Abschnitt 4.1 werden die aus Unterabschnitt 2.3.1 be-

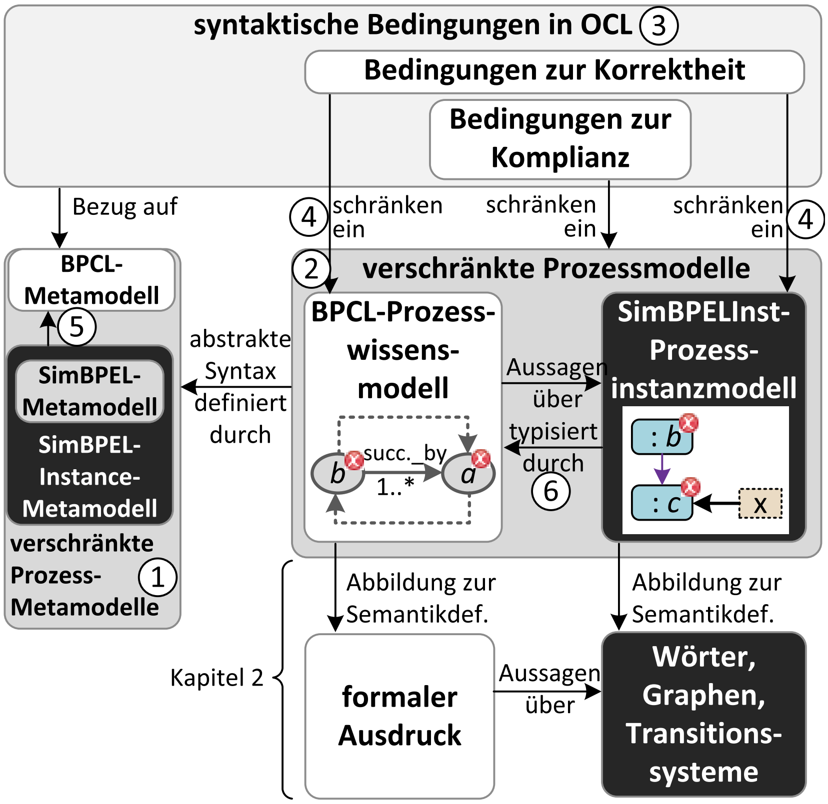


Abbildung 4.1: Zusammenhänge bei der Nutzung expliziten Prozesswissens

kannten Metamodelle ① der in dieser Arbeit verwendeten Prozessmodelle ② aufgegriffen. Die Prozess-Metamodelle definieren jeweils eine Sprache, also eine (unendliche) Menge gültiger Prozesswissen- bzw. Prozessdefinitions- bzw. Prozessinstanzmodelle. Die Ausdrucksmächtigkeit der Prozess-Metamodelle allein ist unzureichend, wie in Abschnitt 4.1 erörtert wird, d.h. in den von ihnen definierten Sprachen befinden sich auch nicht-sinnvolle Prozessmodelle. Die Metamodelle werden daher um textuelle Ausdrücke in der Object Contraint Language erweitert ③ und ihre Ausdrucksstärke soweit erhöht, dass die Menge gültiger Prozessmodelle auf die Menge sinnvoller Prozessmodelle reduziert werden kann (Abschnitt 4.3). Zunächst werden dabei solche OCL-Ausdrücke behandelt, die sich jeweils nur auf ein Metamodell beziehen. Über diese Ausdrücke können Prozessmodelle jeweils syntaktisch so eingeschränkt werden ④, dass ihre Korrektheit gewahrt bleibt und somit technische Probleme vermieden werden können. Fachliche Komplianzbedingungen können im Gegensatz zu technischen Korrektheitsbedingungen nicht an einem Modell allein festgestellt werden, sondern nur anhand zweier Modelle: Ein Prozesswissensmodell als Träger der fachlichen Bedingungen und ein Prozessinstanzmodell als Prüfling für fachliche Bedingungen. Die Realisierung der Komplianzprüfung geschieht wiederum über Bedingungen in der Object Contraint Language. In diesem Fall nehmen die in Abschnitt 4.4

näher betrachten OCL-Ausdrücke Bezug auf ein Gesamtmetamodell, das über Metareferenzen zwischen den einzelnen Prozessmetamodellen etabliert wird ⑤ (Unterabschnitt 4.4.1). Die Metareferenzen finden sich auf Modellebene als Referenzen von Aktivitäten in Prozessinstanzmodellen zu Aktivitätstypen in Prozesswissensmodellen wieder ⑥.

Der Hauptvorteil dieses Ansatzes ist, dass sowohl Korrektheits- als auch Komplianzprüfungen mit einem gemeinsamen, syntaxbasierten Ansatz realisiert werden können.

4.1 Ausdrucksmächtigkeit der Prozess-Metamodelle

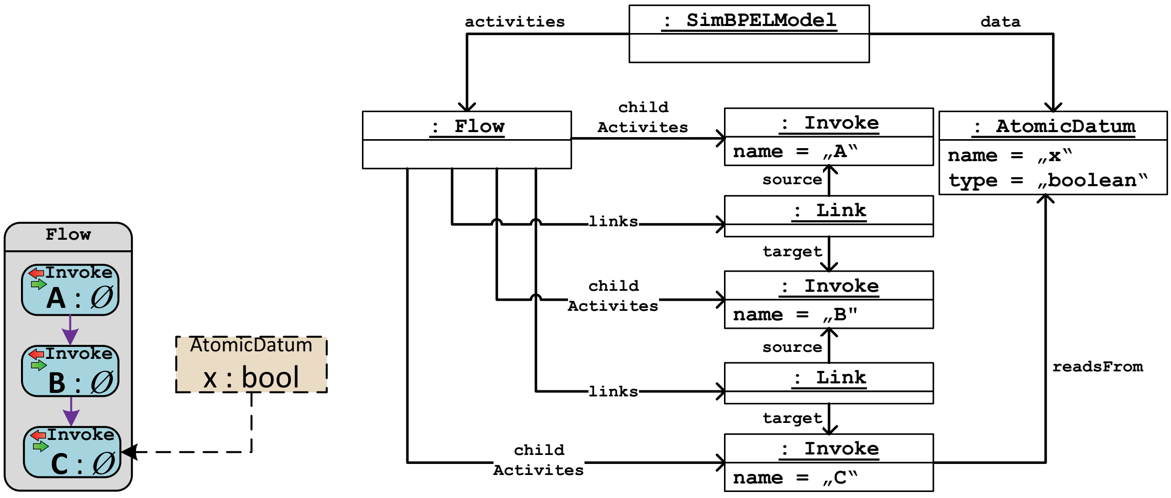
Die Metamodelle der verschiedenen Prozessmodelle wurden in Unterabschnitt 2.3.1 bereits erläutert. Die Sprache, in der die Metamodelle verfasst sind, ist Ecore. In Ecore-Metamodellen können syntaktische Objekte klassifiziert werden und festgelegt werden, welche Objekte untereinander in wievielfacher Beziehung stehen können.

Die Ausdrucksstärke von Ecore reicht nicht aus, um bestimmte Prozessmodelle als syntaktisch unzulässig auszuschießen. Jeder Versuch, durch Änderung des betreffenden Prozess-Metamodells nicht-sinnvolle, aber bislang syntaktisch zulässige Prozessmodelle auszuschließen, führt dazu, dass entweder immer noch nicht-sinnvolle Prozessmodelle nach wie vor syntaktisch zulässig sind oder dass sinnvolle Prozessmodelle syntaktisch unzulässig werden.

Beispiel 4.1 (uninitialisierte Prozessvariable) Bei der Änderung von Prozessinstanzmodellen muss darauf geachtet werden, dass auch nach der Änderung jede Prozessvariable in jedem möglichen Ausführungspfad zunächst geschrieben (initialisiert) und erst dann gelesen wird. Prozessinstanzmodelle wie das in Abbildung 4.2 modellieren insoweit eine nicht-sinnvolle Situation: Hier wird die Prozessvariable x von Aktivität C gelesen ohne vorher mindestens einmal geschrieben worden zu sein.

Es ist in Beispiel 4.1 unerheblich, wie die Situation zustande gekommen ist, d.h.

1. ob sie bereits vorab so im zugehörigen Prozessinstanzmodell modelliert war oder



(a) konkrete Syntax (b) pseudo-abstrakte Syntax

Abbildung 4.2: Lesezugriff auf uninitialisierte Prozessvariable

2. erst durch eine dynamische Änderung im Prozessinstanzmodell entstanden ist. In diesem Fall ist es weiterhin unerheblich, ob die Situation beispielsweise
   a) durch dynamisches Hinzufügen von `C` entstanden ist oder
   b) durch Löschung einer Aktivität, die schreibend auf `x` zugegriffen hat und vor `C` in der Kontrollflussdefinition positioniert war.

Der relevante Teil des Metamodells ist in Abbildung 4.3 dargelegt. Hiervon ist das Prozessinstanzmodell aus Abbildung 4.2(a) ein gültiger Ausdruck. Zu Verdeutlichung der Beziehung zwischen Metamodell und Prozessinstanzmodell ist dieses in Abbildung 4.2(b) in zusätzlich pseudo-abstrakter Syntax abgebildet, d.h. in der generischen konkreten Syntax von UML-Objektdiagrammen.

Das Metamodell schreibt nur vor, dass Beziehungen für Lesezugriffe und Schreibzugriffe (`readsFrom` bzw. `writesTo`) nur zwischen atomaren Aktivitäten (`AtomicActivity`) und Prozessvariablen (`Datum`) existieren können. Hierbei kann aufgrund der Multiplizität `0..*` eine Aktivität auf beliebig viele Prozessvariablen lesend bzw. schreibend zugreifen und eine Prozessvariable von beliebig vielen Aktivitäten gelesen bzw. geschrieben werden.

Modelle wie die aus Abbildung 4.2 sind durch das Metamodell aus Abbildung 4.3 nicht ausgeschlossen. Jede `AtomicActivity` kann auf beliebig viele `Datum`-Elemente zugreifen unabhängig davon, ob andere `AtomicActivities`, die laut Kontrollflussdefinition in jedem Fall vorher durchgeführt werden, die betreffende Prozessvariable schreiben. Notwendig wäre hier eine Kontextbedingung, die die Zulässigkeit einer `readsFrom`-Beziehung in einem Prozessinstanzmodell von einer `AtomicActivity` A zu einem `Datum` P davon abhängig

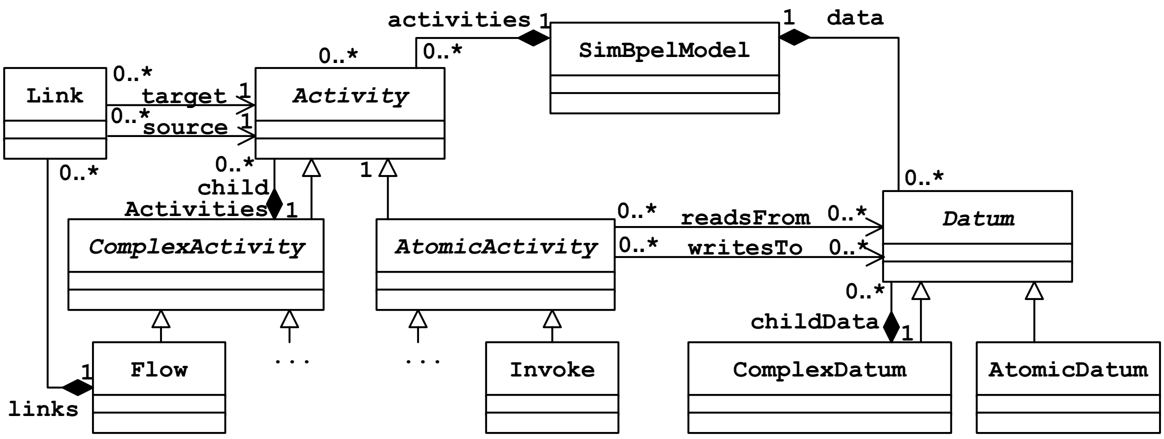


Abbildung 4.3: Teil des SimBPEL-Metamodells

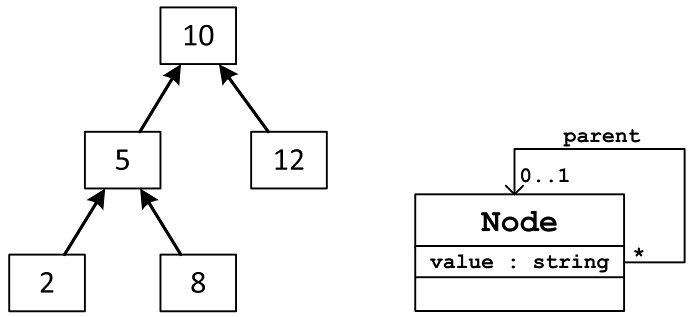
macht, ob eine Aktivität B existiert, die laut Kontrollflussdefinition vor A ausgeführt wird. Auf die Kontrollflussdefinition kann aber in den Metareferenzen `readsFrom` und `writesTo` kein Bezug genommen werden.

4.2 Metamodellerweiterungen um OCL-Ausdrücke

Der Wechsel der Metamodellierungssprache Ecore beispielsweise zu MOF (vgl. Unterabschnitt 2.1.4) bringt keinen Gewinn hinsichtlich der Ausdruckstärke der Metamodelle. Auch in Metamodellen, die in MOF ausgedrückt sind, sind ähnliche Sachverhalte modellierbar wie in Metamodellen, die in Ecore verfasst sind.

Ecore-Metamodelle können durch Ausdrücke der Object Contraint Language (OCL) ergänzt werden. Die Sprachkombination aus Ecore und OCL wird im Folgenden Ecore-OCL genannt; die Ausdrücke hiervon werden Ecore-OCL-Metamodelle genannt. Ecore-OCL-Metamodelle sind ausdrucksstärker als Ecore-Metamodelle.

Die Erhöhung der Ausdrucksstärke von Ecore-Metamodellen durch OCL lässt sich an einem einfachen Beispiel demonstrieren: Es sollen Datenstrukturen, die die Baumeigenschaft aufweisen, diagrammatisch modelliert werden. Jeder Knoten soll dabei einen ganzzahligen Wert tragen. Im Metamodell von Abbildung 4.4(b) ist die abstrakte Syntax von Bäumen so definiert, dass jeder Knoten keinen oder einen eindeutigen Vaterknoten (`parent`) besitzt. In Abbildung 4.4(a) ist eine Modellinstanz in einer einfachen konkreten Syntax dargestellt. Nun sind aber durch das Metamodell folgende Eigenschaften nicht sichergestellt:



(a) Modellinstanz eines Baums

(b) Ecore-Metamodell für Bäume

Abbildung 4.4: (Meta-)Modelle für Bäume

Zykelfreiheit Bäume haben keine Zykel. Laut des Metamodells sind aber auch Zykel in Modellen möglich, d.h. sie werden durch das Metamodell nicht ausgeschlossen.

Zusammenhang Ein Baum ist zusammenhängend, d.h. es gibt eine eindeutige Wurzel. Durch das Metamodell sind aber auch Wälder aus mehreren Bäumen möglich.

Es gibt keine Möglichkeit, das Metamodell so abzuändern, dass nur Modellinstanzen mit den besagten Eigenschaften gültige Modellinstanzen sind. Erst die Ergänzungen aus Listing 4.1 in OCL garantieren diese Eigenschaften.

Der Bezug zwischen OCL-Ausdrücken und Metamodellen wird durch die Angabe eines Kontextes hergestellt. Nach dem Schlüsselwort `context` wird hierbei der Name der Metaklasse angegeben, auf die die folgenden Ausdrücke Bezug nehmen. Alle nachfolgenden Ausdrücke befinden sich im Kontext der angegebenen Metaklasse bis zu der Zeile, in der ein anderer Kontext angegeben wird. Die Evaluation einer Invarianten eines Modellelements m bewirkt insbesondere, dass der Bezeichner `self` in der Invariantendefinition an m gebunden wird. In Modellen werden für jedes Modellelement m jeweils alle Invarianten ausgewertet, die im Kontext (von Superklassen) der Metaklasse angegeben sind. Invarianten werden mit dem Schlüsselwort `inv` gefolgt von einem Invariantennamen angegeben. Nach dem Doppelpunkt folgt die Definition der Invariante.

zykelfrei Im Beispiel von `zykelfrei` ist die Definition wie folgt aufgebaut: Auf `self` kann die Standardfunktion `closure` aufgerufen werden, die auf dem Standardtypen `Set` definiert ist. Der Pfeiloperator `->` bewirkt, dass der linksseitige Ausdruck `self` zu einer Menge mit dem einen Element `self` evaluiert. Der Operation `closure` wird als Parameter den Namen eine Metareferenz übergeben, in diesem Fall `parent`. Sie evaluiert zu einer

```
context Node
inv zykelfrei: not self->closure(parent)->includes(self)

def: istWurzel: Boolean = self.parent->isEmpty()

inv zusammenhaegend: Node.allInstances()->forAll(n |
   n.istWurzel implies Node.allInstances()->forAll(m |
   m.istWurzel implies m = n))
```

Listing 4.1: OCL-Invarianten zur Garantie von Baumeigenschaften

Menge, die den transitiven Abschluss aller `Node`-Elemente beinhaltet, die über `parent`-Referenzen direkt oder indirekt mit `self` verbunden sind. Auf Mengen kann die Mengenoperation `includes` angewendet werden, wobei es sich wiederum um eine Standardfunktion für Mengen handelt. Sie evaluiert zu einem Wahrheitswert und ist genau dann erfüllt, wenn das als Parameter übergebene Element (hier `self`) in der Menge enthalten ist.

istWurzel Neben Invarianten können Operationen definiert werden, die in anderen Operationen oder Invarianten verwendet werden können. Operationen werden durch das Schlüsselwort `def` eingeleitet, gefolgt von einem Operationsnamen, einer Eingabeparameterliste und einem Rückgabewert. Die Operation `istWurzel` ist parameterlos und wahrheitswertig, gibt also ein Wert vom Typ `Boolean` zurück. Nach dem Gleichheitszeichen folgt die Definition der Operation. In diesem Fall wird, wiederum ausgehend vom mit `self` bezeichneten `Node`-Element, mittels Punktoperator über `parent` zum Vater navigiert. Da laut Metamodell ein Knoten einen oder keinen Vater hat, ist `self.parent` mengenwertig, d.h. ist die leere Menge oder eine Menge, die als Element genau den betreffenden Vater enthält. Mittels `isEmpty()` kann getestet werden, ob die Menge leer ist.

zusammenhaengend Der Ausdruck `Node.allInstances()` evaluiert zur Extension von `Node` im betreffenden Modell, d.h. ist eine Menge, die alle Modellelemente der Metaklasse `Node` beinhaltet. Mit `forAll` wird diese Menge allquantifiziert. Hierbei muss für alle `Node`-Elemente `n` gelten, dass, wenn `n` eine Wurzel ist, für alle `Node`-Elemente `m` gilt, dass sie, falls sie auch eine Wurzel sind, identisch mit `n` sind. Hierüber wird ausgeschlossen, dass es zwei unterschiedliche `Node`-Elemente mit Wurzeleigenschaft gibt.

Das Listing 4.1 zeigt alle wesentlichen Ausdrucksmittel von OCL, die im Folgenden verwendet werden. Weitere Ausdrucksmittel die OCL bietet, z.B. zur Beschreibung von Vor- und Nachbedingungen von Operationen, sind für diese Arbeit irrelevant.

```
context AtomicActivity
inv variablesInitialized:
      self.readsFrom->forAll(d |
      AtomicActivity.allInstances()->exists(a |
      a.writesTo->includes(d) and
      self.mandIndirectPreds->includes(a)))
```

Listing 4.2: OCL-Invariante zur Feststellung nicht-initialisierter Variablen [WKH08a]

4.3 Metamodellerweiterungen zur Korrektheitswahrung

Da das Prozessmetamodell aus Abbildung 4.3 unzureichend ist, um Prozessinstanzmodelle wie das aus Abbildung 4.2 syntaktisch auszuschließen, wird es um textuelle Bedingungen in der Object Contraint Language angereichert.

OCL-Ausdrücke können auf Instanzen von Ecore-Metamodellen ausgewertet werden. Insbesondere lassen sich in OCL Invarianten definieren, die in allen Modellen gelten müssen. Wird eine Invariante verletzt, ist das Modell bezogen auf das Metamodell zzgl. der OCL-Ausdrücke syntaktisch nicht korrekt. Ziel der Verwendung von OCL ist daher, genau die Prozessinstanzmodelle syntaktisch auszuschließen, die nicht-sinnvoll sind, beispielsweise das aus Abbildung 4.2.

4.3.1 Korrektheitsbedingungen für Prozessinstanzmodelle

Im Folgenden wird die Umsetzung von Korrektheitsprüfungen in Prozessinstanzmodellen erläutert, die auch auf Prozessdefinitionsmodelle anwendbar sind, da diese lediglich Spezialfälle von Prozessinstanzmodellen sind.

Beispiel 4.2 (Uninitialisierte Prozessvariablen) In Beispiel 4.2 wurden uninitialisierte Prozessvariablen bereits als nicht-sinnvolle Situationen in Prozessinstanzmodellen beschrieben. In Listing 4.2 ist eine Invariante `variablesInitialized` in OCL definiert, die derartige Situationen in Prozessinstanzmodellen in OCL formalisiert. Diese Invariante wird auf Modellelemente der Metaklasse `AtomicActivity` ausgewertet und evaluiert zu einem Wahrheitswert (`true` oder `false`). Die betreffende `AtomicActivity` wird in der Definition der Invariante mit `self` referenziert. Die Definition

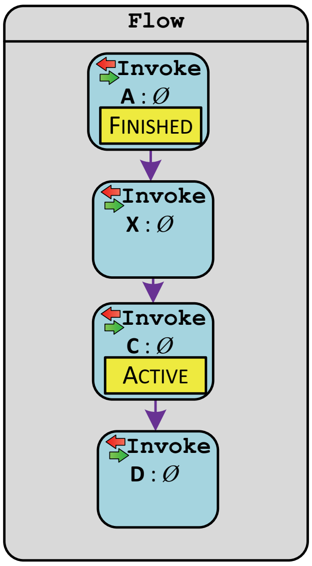


Abbildung 4.5: Prozessinstanzmodell mit unerreichbarer Aktivität X

ist wie folgt zu lesen: Für alle (`forAll`) Variablen `d` (Metaklasse `Datum`), auf die `self` lesend (`readsFrom`) zugreift, muss gelten: Es muss in der Extension aller atomaren Aktivitäten (`AtomicActivity.allInstances()`) eine atomare Aktivität `a` existieren (`exists`), die schreibend auf `d` zugreift (`a.writesTo->includes(d)`) und in der Menge `self.mandIndirectPreds` (vgl. Definition 4.1) der in jedem Fall vor `self` durchgeführten Aktivitäten enthalten ist.

Die Invariante in Listing 4.2 hat drei Anwendungsstellen im Prozessinstanzmodell von Abbildung 4.2, da die `Invoke`-Aktivitäten A, B und C Modellelemente der Metaklasse `Invoke` sind und aufgrund der Vererbungsbeziehung im Metamodell auch von `AtomicActivity`. Aktivitäten A und B lesen von keiner Prozessvariablen. Die Allquantifizierung über die jeweils leere Menge `self.readsFrom` ist daher trivialerweise erfüllt. Im Fall von C (d.h. `self = c`) ist allerdings x in der Menge `self.readsFrom` und A und B in der Menge `self.mandIndirectPreds`. Da weder A noch B schreibend auf eine Variable zugreifen, `a.writesTo` also leer ist und `a.writesTo->includes(d)` somit für beliebiges `d` nicht erfüllt ist, ist der Gesamtausdruck in diesem Fall nicht erfüllt.

Beispiel 4.3 (Unerreichbare Aktivitäten) Nicht sinnvoll sind auch solche Prozessinstanzmodelle, die unerreichbare Aktivitäten beinhalten. Zwar

```
context Activity

def: isInAliveState: Boolean =
  self.oclIsKindOf(Invoke) and
  (not self.oclAsType(Invoke).activityInstances->isEmpty()
  implies self.oclAsType(Invoke).activityInstances->exists(i |
  i.state.isWaiting or i.state.isActive))

def: optIndirectSuccOfAliveAct: Boolean =
  Invoke.allInstances()->exists(i | i.isInAliveState and
  i.optIndirectSuccs->includes(self))

inv invReachable:isInAliveState implies optIndirectSuccOfAliveAct
```

Listing 4.3: OCL-Invariante zur Erreichbarkeitsprüfung von Aktivitäten

führen unerreichbare Aktivitäten nicht zu technischen Fehlern, sind jedoch unabhängig von spezifischem Fachwissen in allen Situationen nicht gewollt. Daher werden Prozessinstanzmodelle mit unerreichbaren Aktivitäten als syntaktisch inkorrekt betrachtet.

In Listing 4.3 ist eine OCL-Invariante `invReachable` spezifiziert, die sich auf zwei wahrheitswertige Funktionen `isInAliveState` und `optIndirectSuccOfAliveAct` abstützt.

isInAliveState Die Funktion `isInAliveState` berechnet für eine `Invoke`-Aktivität, ob diese in einem lebendigen Zustand ist. Dies ist der Fall, wenn sie entweder keine Aktivitätsinstanz hat (`ActivityInstance`) oder eine Aktivitätsinstanz in einem wartenden (`isWaiting`) oder aktiven Zustand (`isActive`) besitzt.

optIndirectSuccOfAliveAct Diese Funktion berechnet, ob die fragliche Aktivität (`self`) innerhalb der möglichen Nachfolger (`optIndirectSuccs`) einer lebendigen Aktivität (`isInAliveState`) ist.

invReachable Die Invariante ist erfüllt, wenn die fragliche Aktivität selbst entweder nicht (mehr) lebendig ist oder falls doch, dann auch Nachfolger einer anderen lebendigen Aktivität ist.

Die Invariante `invReachablility` ist für Aktivität X aus Abbildung 4.5 nicht erfüllt. X selbst ist in einem lebendigen Zustand. Es gibt jedoch keine weitere Aktivität im Prozessinstanzmodell, die sowohl selbst lebendig ist als auch X als mögliche Nachfolgeraktivität besitzt.

Kontrollflussabhängigkeiten

Bei der Auswertungen der OCL-Invarianten werden, zumindest mittelbar, OCL-Operationen angewendet, die die gegenseitigen Kontrollflusspositionen der im Prozessinstanzmodell enthaltenen Aktivitäten berechnen. In Beispiel 4.2 und Beispiel 4.3 werden die Mengen der obligatorischen Vorgänger oder der optionalen Nachfolger über die Operationen `mandIndirectPreds` bzw. `optIndirectSuccs` berechnet.

Die Berechnung dieser Mengen ist nicht trivial. Der Grund dafür ist, dass sich Kontrollflussbeziehung zwischen Aktivitäten zum Teil unmittelbar aus den vorhandenen syntaktischen Elementen der Prozessinstanzmodelle ergeben. Manche Kontrollflussbeziehungen bestehen aufgrund bestimmter komplexer Konstellationen in der syntaktischen Struktur des Modells. Bei einer Auswertung der Syntax mittels OCL-Ausdrücken muss dabei insbesondere auf die Semantik der verschiedenen Modellelemente Rücksicht genommen werden. Beispielsweise ist zu beachten, dass aufgrund von Schleifen eine Aktivität A sowohl Vorgänger als auch Nachfolger einer anderen Aktivität B sein kann.

Im Folgenden wird die Kontrollflussbeziehung zwischen zwei Aktivitäten schrittweise über OCL-Operationen definiert. Die OCL-Operationen berechnen dabei Mengen, die für eine Aktivität a alle indirekten Vor- oder Nachfolgeraktivitäten beinhalten. Hierbei ist jeweils zwischen mandatorischen und optionalen Aktivitäten zu unterscheiden. Erstere werden im jedem Fall vor bzw. nach a durchgeführt, letztere nur in mindestens einem Ausführungspfad. Es ergeben sich also vier Mengen, die berechnet werden müssen. Ausgangspunkt ist jeweils eine formale Definition der jeweiligen Menge.

Diese Definition bezieht sich einerseits auf die mit Akt bezeichnete Menge aller Aktivitäten im Prozessinstanzmodell, andererseits auf das Graphtransitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel},\mathrm{pg}(p))}$ (vgl. Abschnitt 2.5) und abstrahiert somit von der Syntax der Prozessinstanzmodelle. Die generelle Form der formalen Definition der Mengen ist

$$\mathcal{M}_p(b) = \{a \in Akt(p) | \varphi(a, b)\}.$$

Hierbei sind a und b Aktivitäten aus der Aktivitätenmenge Akt im Prozessinstanzmodell p und φ eine CTL-Formel, in der a und b als atomare Aussage vorkommen. Die Formel wird auf dem Graphtransitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel},\mathrm{pg}(p))}$ ausgewertet (vgl. Unterabschnitt 2.5.3). Vereinfachend wird hierbei der Aktivitätstyp an die Stelle eines eindeutigen Aktivitätsbezeichners gesetzt.

Mandatorische Kontrollflussabhängigkeiten Eine Aktivität a ist mandatorisch kontrollflussabhängig von einer anderen Aktivität b, wenn b in jedem Fall, also jedem möglichen Ausführungspfad, vor bzw. nach a ausgeführt wird.

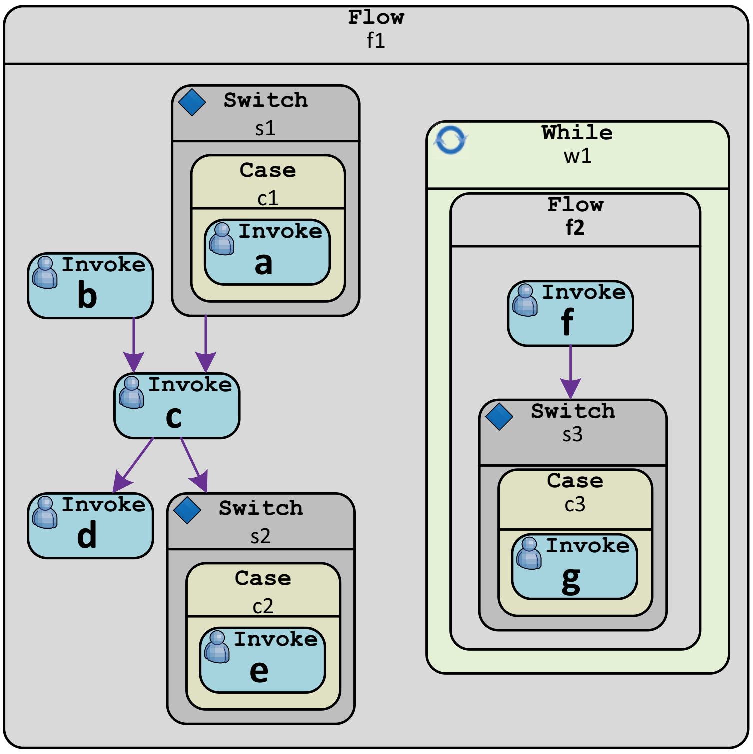


Abbildung 4.6: Beispielhaftes Prozessinstanzmodell p zur Erläuterung von Kontrollflussbeziehungen

Definition 4.1 (mandatorische Vorgänger)
Die Menge $\text{mandIndirectPreds}_p(b)$ der mandatorischen Vorgänger einer Aktivität b in einem Prozessinstanzmodell p ist definiert als

$$\text{mandIndirectPreds}_p(b) := \{a \in Akt(p) | \neg \mathbf{E}(\neg a \mathbf{U} b)\}.$$

In der Menge sind also genau die Aktivitäten a enthalten, für die gilt: Es gibt keinen Ausführungspfad auf dem b ausgeführt wird und vorher niemals a.

Im Beispiel von Abbildung 4.6, die das Prozessinstanzmodell P zeigt, ist $\text{mandIndirectPreds}_p(c) = \{b\}$. Die Aktivitäten a, f und g werden nur möglicherweise vor c ausgeführt, die Aktivitäten d und e in keinem Fall.

Definition 4.2 (mandatorische Nachfolger)
Die Menge $\text{mandIndirectSuccs}_p(b)$ der mandatorischen Nachfolger einer Aktivität b in einem Prozessinstanzmodell p ist definiert als

$$\text{mandIndirectSuccs}_p(b) := \{a \in Akt(p) | \mathbf{AF}(b \Rightarrow \mathbf{AF}a)\}.$$

In der Menge befinden sich genau die Aktivitäten a für die gilt: Sobald b ausgeführt wurde, muss in jedem Fall in der Zukunft (**AF**) auch a ausgeführt werden.

Bezogen auf das Beispiel in Abbildung 4.6 ist $\text{mandIndirectSuccs}_p(\mathsf{c}) = \{\mathsf{d}\}$. Die anderen Aktivitäten werden entweder nie oder nur in manchen Fällen nach c ausgeführt.

Optionale Kontrollflussabhängigkeiten Mitunter sind bei der Formulierung von Invarianten zur Korrektheitswahrung auch die Mengen der optional kontrollflussabhängigen Aktivitäten relevant. Dies trifft beispielsweise auf die Invariante zur Erreichbarkeit in Listing 4.3 zu, in der die Menge `optIndirectSuccs` der optionalen Nachfolger einer (an `i` gebundenen) Aktivität verwendet wird. Eine Aktivität a ist optional kontrollflussabhängig von einer anderen Aktivität b, wenn b in mindestens einem Fall, also einem möglichen Ausführungspfad, vor bzw. nach a ausgeführt wird. Auch hier wird wieder zwischen Mengen für Vorgänger- und Nachfolgeraktivitäten unterschieden.

Definition 4.3 (optionale Vorgänger) Die Menge $\text{optIndirectPreds}_p(b)$ der optionalen Vorgänger einer Aktivität b in einem Prozessinstanzmodell p ist definiert als

$$\text{optIndirectPreds}_p(b) := \{a \in Akt(p) | \mathbf{EF}(a \wedge \mathbf{EF}b)\}.$$

In dieser Menge sind genau die Aktivitäten a enthalten, für die ein Ausführungspfad existiert, auf dem a und danach b ausgeführt wird.

Im Beispiel von Abbildung 4.6 ist $\text{optIndirectPreds}_p(\mathsf{c}) = \{\mathsf{a,b,f,g}\}$. Die Aktivitäten d und e sind nicht enthalten, weil sie in keinem Fall vor c durchgeführt werden.

Definition 4.4 (optionale Nachfolger)
Die Menge $\text{optIndirectSuccs}_p(b)$ der optionalen Nachfolger einer Aktivität b in einem Prozessinstanzmodell p ist definiert als

$$\text{optIndirectSuccs}_p(b) := \{a \in Akt(p) | \mathbf{EF}(b \wedge \mathbf{EF}a)\}.$$

In dieser Menge sind genau die Aktivitäten a enthalten, für die ein Ausführungspfad existiert, auf dem b und danach a ausgeführt wird.

In Abbildung 4.6 ist $\text{optIndirectSuccs}_p(\mathsf{c}) = \{\mathsf{d},\mathsf{e},\mathsf{f},\mathsf{g}\}$. Die Aktivitäten `a` und `b` sind nicht enthalten, weil sie in keinem Fall nach `c` durchgeführt werden.

Syntaxbasierte Berechnung der Kontrollflussbeziehung Die vorgehenden Definitionen der Kontrollflussbeziehungen können nicht direkt in eine implementierte Berechnung überführt werden. Sie beziehen sich auf das Graphtransitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel},\text{pg}(p))}$, das sich durch Anwendung der Graphersetzungsregeln $\mathcal{R}_{bpel}$ aus Unterabschnitt 2.5.3 auf den Prozessgraphen $\text{pg}(p)$ eines Prozessinstanzmodells p ergibt. Demgegenüber beziehen sich OCL-Ausdrücke immer auf die Syntax der Prozessmodelle. Die auf CTL basierende Definition der Mengen muss daher in eine Reihe von OCL-Ausdrücken überführt werden. In diesem Unterabschnitt soll dies exemplarisch anhand von `mandIndirectSuccs` gezeigt werden.

Der erste Schritt bei der Berechnung von `mandIndirectSuccs` ist die Berechnung der direkten Nachfolger in der syntaktischen Struktur des Prozessinstanzmodells. Hierbei ist zu beachten, dass die direkte Nachfolgerschaft über mehrere syntaktische Elemente bzw. bestimmt Kombinationen syntaktischer Elemente in Prozessinstanzmodellen erreicht werden kann. Direkte Nachfolgerschaft ist hier so zu verstehen, dass ein direkter Nachfolger b von a nicht unbedingt auch direkt nach a ausgeführt werden muss, jedoch über eine bestimmte syntaktische Beziehung in einem Schritt in der Prozessmodellsyntax von a aus erreicht werden kann.

Aus Listing 4.4 ist ersichtlich, dass die direkte Nachfolgerschaft über mehrere syntaktische Konstellationen zustande kommen kann, die jeweils durch eine eigene OCL-Operation abgedeckt werden müssen. Ihre Definitionen bauen teils aufeinander auf und sind wie folgt zu lesen:

directLinkSuccs Diese Operation berechnet für eine Aktivität die Menge der durch `Links` verbundenen direkten Nachfolger. Angewendet auf Aktivität `c` in Abbildung 4.6 ist dies Aktivität `d` und `s2`.

directChildren liefert direkte Kinder einer komplexen Aktivität.

directAncestorSuccs berechnet für eine Aktivität die Menge der direkten `Link`-Nachfolger der ggf. vorhandenen komplexen Elternaktivität. Ist diese Menge leer, wird die Operation rekursiv auf die Elternaktivität angewendet. Nur im Fall, dass sämtliche Vorfahren keine direkten `Link`-Nachfolger haben, liefert die Operation letztendlich die leere Menge. Wendet man die Operation auf Aktivität `a` an, so wird aufgrund des Vorfahrens `s1` eine Menge mit `c` als einzigem Element zurückgegeben.

directSuccs vereinigt die drei zuvor definierten Mengen.

```
context Activity

def: directLinkSuccs: Set(Activity) =
 Activity.allInstances()->select(act: Activity |
 Link.allInstances()->exists(link: Link |
 link.source = self and link.target = act))

def: directChildren: Set(Activity) =
 if self.oclIsKindOf(ComplexActivity) then
  self.oclAsType(ComplexActivity).childActivities
 else Set{} endif

def: directAncestorSuccs: Set(Activity) =
 if parentActivity <> null then
  if parentActivity.directLinkSuccs->isEmpty() then
   parentActivity.directAncestorSuccs
  else parentActivity.directLinkSuccs
 endif

def: directSuccs: Set(Activity) =
 -- Vereinigung der durch directLinkSuccs, directChildren,
 -- directASuccs und directWhileSuccs definierten Mengen
 -- (aus Platzgründen ausgelassen)
```

Listing 4.4: OCL-Operationen zur Berechnung direkter Nachfolgerschaften

Die Menge `directSuccs` ist allein nicht verwendbar. Erstens beinhaltet sie nur direkte Nachfolger, zweitens auch optional durchgeführte Aktivitäten. In Listing 4.5 werden daher durch die Operation `indirectSuccs` alle indirekten Nachfolger einer Aktivität berechnet. Hierzu wird der transitive Abschluss mittels der Standardoperation `closure` über die direkten Nachfolger `directSuccs` gebildet. Aus dieser Menge werden in `mandIndirectSuccs` schließlich noch diejenigen Aktivitäten entfernt, die Nachfahren von `Switch`- oder `While`-Aktivitäten sind und daher nur optional ausgeführt werden.

Die Listings zuvor zeigen, dass es möglich ist, Mengen von Aktivitäten, die sich auf den Verhaltensaspekt eines Prozessinstanzmodells beziehen wie die Kontrollflussabhängigkeit zwischen Aktivitäten, allein anhand der syntaktischen Struktur eines Prozessinstanzmodells zu berechnen. Die Berechnung der anderen Mengen verläuft analog.

```
context Activity

def: indirectSuccs: Set(Activity) = self->closure(directSuccs)

def: mandIndirectSuccs: Set(Activity) =
  indirectSuccs - indirectSuccs->select(a |
    a.ancestors->exists(s |
      (s.oclIsTypeOf(Switch) or s.oclIsTypeOf(While))
      and not self.ancestors->includes(s)))
```

Listing 4.5: OCL-Operationen zur Berechnung indirekter mandatorischer Nachfolgerschaften

4.3.2 Korrektheitsbedingungen für Prozesswissensmodelle

Korrektheitsbedingungen, die sich in Metamodellen selbst nicht formulieren lassen, existieren auch für Prozesswissensmodelle. In diesen Modellen lassen sich fachliche Forderungen definieren, die einander ausschließen.

Beispiel 4.4 (Widersprüchliche Reihenfolgebedingungen) In Abbildung 4.7 ist ein syntaktisch inkorrektes Prozesswissensmodell abgebildet. Der Fehler liegt hierbei in der widersprüchlichen, weil zyklischen Festlegung der preceded_by-Bedingungsbeziehung.

Der Widerspruch ergibt sich aus der CTL-Semantik der Bedingungsbeziehungen. Beide formulierte Bedingungen müssen gleichzeitig gelten. Zusätzlich gilt die Nebenbedingung, dass in einem Zustand immer nur eine atomare Aussage gelten kann, was sich aus der Graphgrammatik $\mathcal{G}_{\text{bpel}}$ ergibt. Insbesondere kann also in keinem Zustand $a \wedge b$ gelten. Alle drei Bedingungen entsprechen der CTL-Formel

$$\neg\mathbf{E}(\neg a\mathbf{U}b) \wedge \neg\mathbf{E}(\neg b\mathbf{U}a) \wedge \neg\mathbf{EF}(a \wedge b).$$

Die Formel ist nur erfüllbar, wen niemals *a* oder *b* gilt, da die erste Klausel fordert, dass *a* nicht echt vor *b* gelten darf, die zweite, dass *b* nicht echt vor *a* gelten darf. Infolgedessen darf also weder *a* noch *b* alleinig zuerst gelten. Ein Zustand, in dem *a* und *b* gleichzeitig gelten, ist durch die dritte Klausel ausgeschlossen.

Es ist vernünftig anzunehmen, dass ein Prozesswissensmodellierer den Ausschluss von *a*- und *b*-Aktivitäten aus einem *p*-Prozess auf direktem Wege

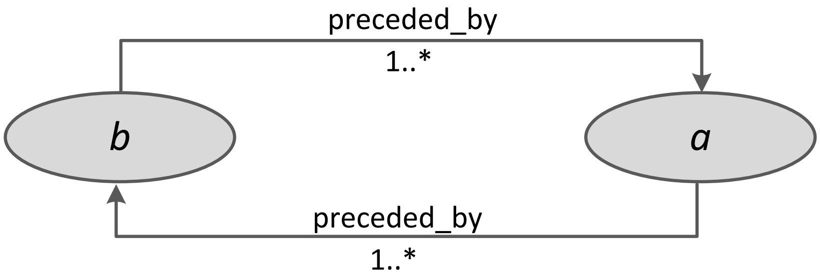


Abbildung 4.7: Inkorrektes Prozesswissensmodell

```
context ActivityType

def: adjRCEs(rcKind: RelConKind) : Set(RelConKind) =
      self.getAllParentTypes.fromSource->select(
      constraintKind = rcKind)

def: directPrecededBy: Set(ActivityType) =
  self.adjRCEs(RelConKind::PRECEDENCE)->select(r |
  r.lowerBound >= 1)->collect(target)->asSet()

inv invNoPrecededByCycle:
   not self->closure(directPrecededBy)->includes(self)
```

Listing 4.6: OCL-Operationen und -Invariante zum Ausschluss von widersprüchlichen preceded_by-Bedingungsbeziehungen

über die Bedingungsbeziehung $p \xrightarrow[0..0]{includes} a$ und $p \xrightarrow[0..0]{includes} b$ modellieren würde, jedoch nicht indirekt wie in Abbildung 4.7.

Wie bei Prozessinstanzmodellen ist daher auch bei Prozesswissensmodellen das Ziel, derartige Situationen als syntaktisch falsch zu erkennen. Auch hier werden wiederum OCL-Invarianten verwendet, die bei derartigen Situationen verletzt sind. In Listing 4.6 sind die OCL-Ausdrücke aufgeführt, die zyklische preceded_by-Beziehungen ausschließen. Hierbei werden für einen Aktivitätstyp `self` über `adjRCEs` und `directPrecededBy` diejenigen Aktivitätstypen bestimmt, die über eine preceded_by-Bedingungsbeziehung mit unterer Schranke (`lowerBound`) von mindestens 1 verbunden sind. In der Definition der Invariante von `invNoPrecededByCycle` wird geprüft, ob `self` im transitiven Abschluss der durch `directPrecededBy` gebildeten abgeleiteten Beziehung ist, also ein durch diese Beziehung gebildeter Zykel vorliegt.

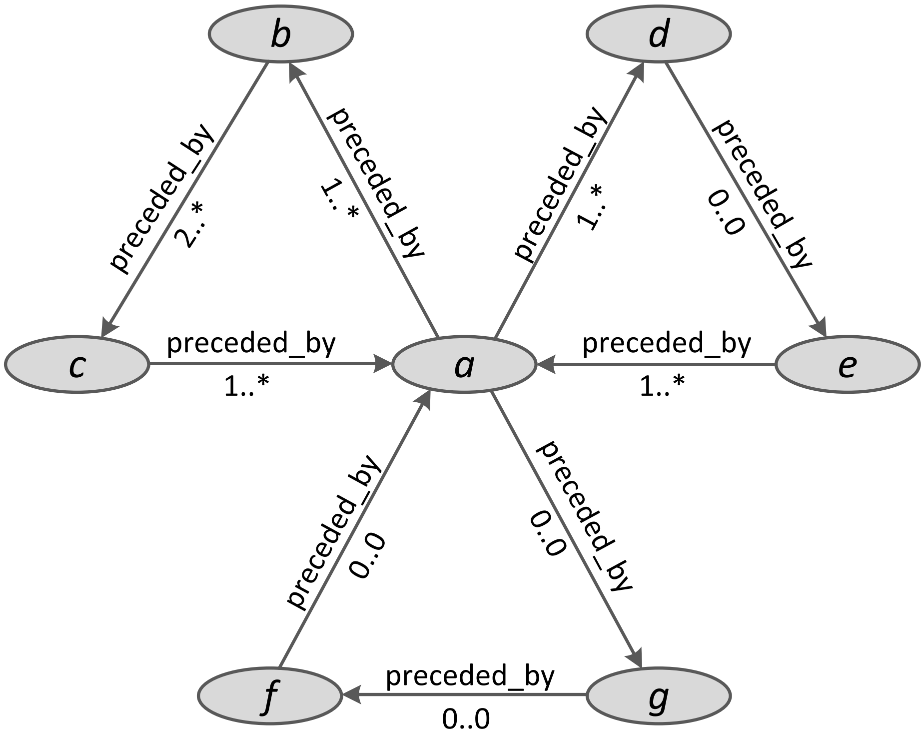


Abbildung 4.8: Teilweise inkorrektes Prozesswissensmodell

Beispiel 4.5 (Verletzung und Einhaltung von `invNoPrecededByCycle`)
Die Invariante `invNoPrecededByCycle` ist für die Aktivitätstypen *a*, *b* und *c* aus Abbildung 4.8 nicht erfüllt, da diese so einen Zykel preceded_by-Bedingungsbeziehungen bilden, dass keine Aktivität als erstes ausgeführt werden darf. Demgegenüber ist `invNoPrecededByCycle` für die Aktivitätstypen *d* und *e* erfüllt. Sie bilden zusammen mit *a* keinen Zykel, da $d \xrightarrow[0..0]{preceded\_by} e$ fordert, dass *keine* *e*-Aktivität vor einer *d*-Aktivität ausgeführt werden darf. Eine Ausführungsreihenfolge *d*,*a*,*e* ist also zulässig.

Auf analoge Weise werden drei weitere Invarianten gebildet: Die Invariante `invNoUnprecededByCycle` prüft, ob ein Zykel mit negativer Präzedenzforderung vorliegt. Dies ist beispielsweise im Zykel *a*, *f*, *g* aus Abbildung 4.8 der Fall. Hier wird gefordert, dass vor einer *a*-Aktivität maximal 0, also keine, *g*-Aktivität ausgeführt werden darf, vor einer *g*-Aktivität keine *f*-Aktivität und vor einer *f*-Aktivität keine *a*-Aktivität. Diese Forderungen sind in Kombination unerfüllbar. Analog zu `invPrecededByCycle` und `invNoUnprecededByCycle` werden die Invarianten `invNoSucceededByCycle` und `invNoUnsucceededByCycle` definiert für succeeded_by-Bedingungsbeziehungen.

4.4 Metamodellerweiterungen zur Komplianzwahrung

Der vorhergehende Abschnitt hat gezeigt, dass durch in OCL formulierte Invarianten solche Prozessmodelle als inkorrekt identifiziert werden können, die domänenunabhängige, also allgemeingültige Bedingungen an Prozessmodelle verletzen. Die Einhaltung dieser Bedingungen vermeidet technische Probleme wie nicht ausführbare Prozessinstanzmodelle oder widersprüchliche Prozesswissensmodelle. In der Realisierung wirkt sich die Allgemeingültigkeit der Korrektheitsbedingung so aus, dass sich die entsprechenden OCL-Ausdrücke immer nur auf ein Metamodell beziehen, also beispielsweise ein Prozesswissensmodell oder ein Prozessinstanzmodell.

Dieser Abschnitt befasst sich mit der Verbindung von Prozesswissensmodellen, in denen fachliche Komplianzbedingungen an Prozesse modelliert werden können und Prozessinstanzmodellen, in denen diese Bedingungen eingehalten oder verletzt werden. Wiederum erfolgt die Definition der syntaktischen Einschränkungen über OCL. Im Unterschied zu den OCL-Bedingungen zur Prüfung der Korrektheit aus Abschnitt 4.3 nehmen die OCL-Bedingungen zur Prüfung der Komplianz jeweils Bezug auf Prozessinstanzmodelle *und* -wissensmodelle. Das heißt, dass aus je einem Prozessinstanzmodell und einem Prozesswissensmodell zunächst ein zusammenhängendes syntaktische Objekt gebildet werden muss, was in Unterabschnitt 4.4.1 beschrieben ist. Die OCL-Bedingungen werden dann an diesem zusammenhängenden Modell ausgewertet; Einzelheiten hierzu sind in Unterabschnitt 4.4.2 beschrieben.

4.4.1 Prozessmodellverschränkung

Die Möglichkeit zur Verschränkung zweier verschiedenartiger Modelle zu einem zusammenhängenden Modell muss in den betreffenden Metamodellen berücksichtigt werden. In Abbildung 2.12 sind mehrere Verschränkungen aufgezeigt.

Zunächst ist das Metamodell für Prozessinstanzmodelle (SimBPEL-Instanz-Metamodell) lediglich eine Erweiterung des Metamodells für Prozessdefinitionsmodelle (SimBPEL-Metamodell), da Prozessinstanzmodelle in dieser Arbeit als Prozessdefinitionsmodelle zuzüglich der Zustandsinformationen eines bestimmten Prozesses aufgefasst werden.

Wesentlich für Komplianzprüfung ist jedoch die durch die Metareferenz `typedAs` etablierte Verschränkung des SimBPEL-Metamodells (für Prozess-

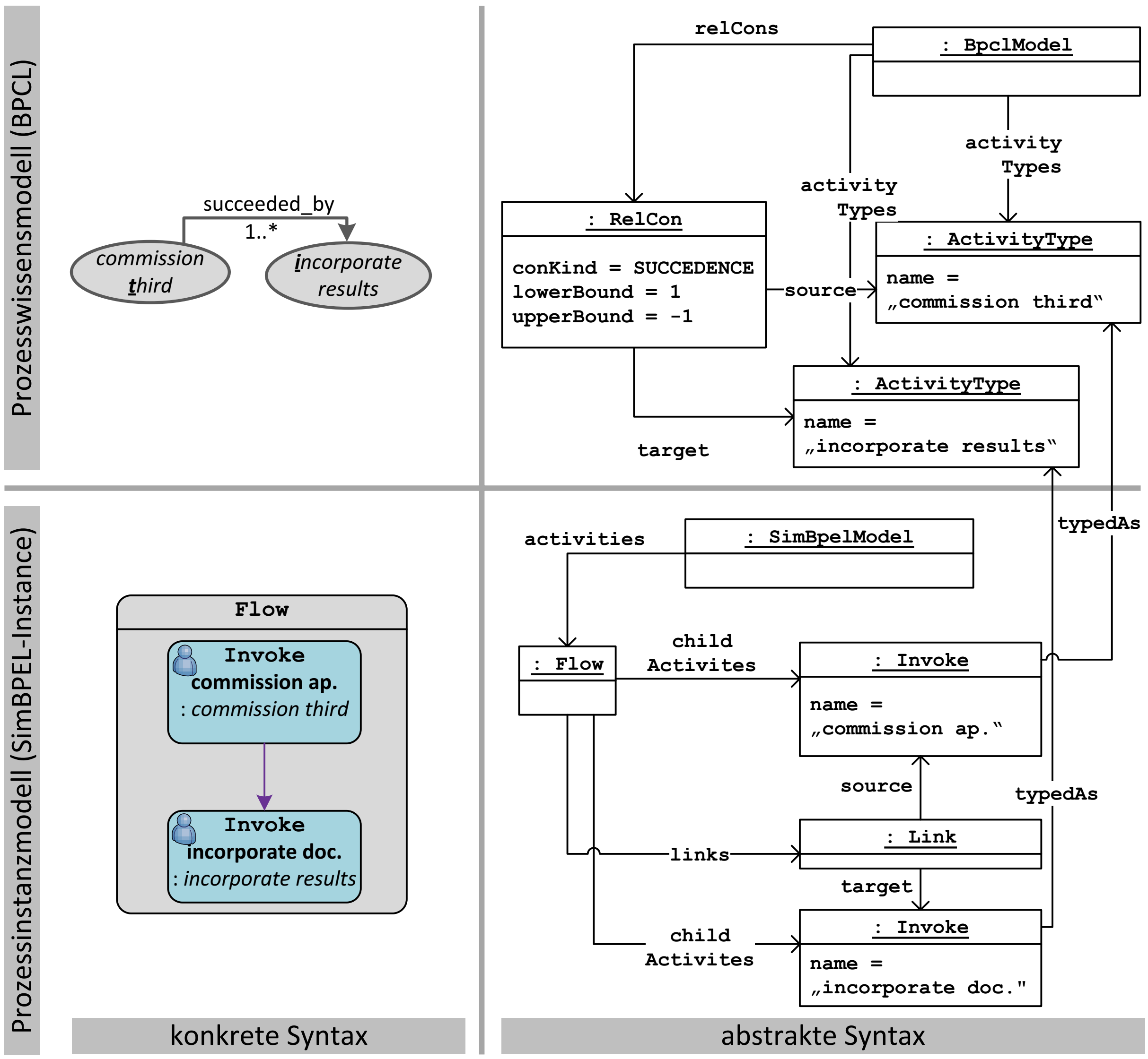


Abbildung 4.9: Beispiel zweier verschränkter Prozessmodelle

definitionsmodelle) und des BPCL-Metamodells (für Prozesswissensmodelle). Diese Metareferenz ermöglicht, dass Aktivitäten in Prozessdefinitions- und somit auch -instanzmodellen Bezug auf Prozesswissen nehmen, das in einem Prozesswissensmodell spezifiziert ist.

Beispiel 4.6 (Verschränkte Prozessmodelle) In Abbildung 4.9 ist ein Prozessinstanzmodell abgebildet, das mit einem Prozesswissensmodell verschränkt ist. In der abstrakten Syntax der Modelle wird die Verschränkung durch `typedAs`-Referenzen hergestellt. Über diese ist festgelegt, dass das `Invoke`-Element mit Namen (`name`) `commission ap.` auf den Aktivitätstypen mit Namen `commission third` verweist, das `Invoke`-Element mit Namen `incorporate doc.` auf den Aktivitätstypen mit Namen `incorporate results`.

Es wäre unpraktisch, Modellinstanzen der Metareferenz `typedAs` in der konkreten Syntax ein visuelles Element – beispielsweise ein Pfeil – zuzuordnen. In diesem Falle müsste im Prozessmodelleditor zu einem Prozessinstanzmodell immer das entsprechende Prozesswissensmodell geöffnet werden. Stattdessen findet sich eine `typedAs`-Referenz im Prozessinstanzmodell nur indirekt wieder. Ist einer Aktivität A ein Aktivitätstyp T zugeordnet, so findet sich der Name des Aktivitätstyps in der Beschriftung der Aktivität wieder und zwar nach dem Aktivitätsnamen, abgetrennt durch einen Doppelpunkt. Diese Notation ist UML-Objektdiagrammen entlehnt, in denen einem Objektbezeichner der Name der Klasse folgt, von dem das betreffende Objekt instanziiert wurde.

4.4.2 OCL-Invarianten für verschränkte Prozessmodelle

In Listing 4.7 ist die OCL-Invariante aufgelistet, in der der Bezug zwischen Aktivitäten in Prozessinstanzmodellen und succeeded_by-Bedingungsbeziehungen in Prozesswissensmodellen hergestellt wird. Sie ist für genau die Aktivitäten in Prozessinstanzmodellen nicht erfüllt, die mit einem Aktivitätstyp getypt sind, der im Prozesswissensmodell eine succeeded_by-Bedingungsbeziehung besitzt, also Quelle einer solchen Beziehung ist, die im Prozessinstanzmodell nicht erfüllt wird. Im Beispiel von Abbildung 4.9 ist die Invariante für beide Aktivitäten im Prozessinstanzmodell erfüllt.

Die OCL-Invariante `invConformsSuccedence` aus Listing 4.7 stützt sich auf diverse Operationen ab, deren Definition hier nur informal erläutert wird. Der Aufbau der Invariante selbst ist wie folgt und typisch auch für die Invarianten anderer Arten von Bedingungsbeziehungen (vgl. Unterabschnitt 2.2.3). In der Invariante wird zunächst geprüft, ob die betreffende (an `self` gebundene) Aktivität überhaupt getypt ist. Falls ja, kann der Typ über `self.typedAs` ermittelt werden; hier kommt die Verschränkung der Prozessmodelle zum Tragen. Für diesen Typ werden über `adjRCEs(RelConKind::SUCCEEDENCE)` alle ausgehenden succeeded_by-Bedingungsbeziehungen `r` gesammelt und jeweils über `matchUpperBound` und `matchLowerBound` überprüft, ob die angegebenen Multiplizitäten im Prozessinstanzmodell eingehalten werden. Hierbei zählen `countMaxTypeSucc` und `countMinTypeSucc` die maximal bzw. minimal mögliche Anzahl der Aktivitätsdurchführungen von Aktivitäten mit Typ `r.target`, die im Kontrollfluss Nachfolger von `self` sind. Beide Werte weichen dann voneinander ab, wenn Verzweigungen (`Switch`-Aktivitäten) oder Schleifen

(While-Aktivitäten) laut Kontrollflussdefinition nach self kommen.

Beispiel 4.7 (Anwendung von invConformsSuccedence)
Die OCL-Invariante `invConformsSuccedence` ist erfüllt, wenn sie auf die Aktivität commission ap. aus Abbildung 4.9 angewendet wird. Der Ausdruck `self.typedAs` evaluiert in diesem Fall zum Aktivitätstyp mit Namen (name) `commission third`, wie aus der abstrakten Syntax von Abbildung 4.9 hervorgeht. Die Anwendung `adjRCEs` liefert das einzige `RelCon`-Element in diesem Modell, d.h. dass `forAll` über genau ein Element `r` quantifiziert, für das `r.target` den Aktivitätstypen mit Namen `incorporate results` liefert. Die Operationsanwendungen `self.countMaxTypeSucc(r.target)` und `self.countMinTypeSucc(r.target)` liefern in diesem einfachen Prozessinstanzmodell beide einen Wert von 1. Dieser liegt in der Multiplizitätsangabe `1..-1`, wobei `-1` die Ganzzahldarstellung für “positiv unendlich” ist, welches sonst als $*$ dargestellt wird; insofern evaluieren die Operationen `matchUpperBound` und `matchLowerBound` zu `true`, so dass die Invariante insgesamt erfüllt ist.

4.5 Prototypische Realisierung

Die in den Abschnitten zuvor beschriebenen Konzepte wurden im Prozessmodelleditor implementiert, so dass Prozessbeteiligte dynamische Änderungen in Prozessinstanzmodellen zunächst auf Korrektheit und Komplianz hin prüfen lassen können, bevor sie an eine Laufzeitumgebung wie den WebSphere Process Server propagiert werden. Unterabschnitt 4.5.1 beschreibt die hierzu gehörigen Funktionen und Sichten des Prozessmodelleditors; in Unterabschnitt 4.5.2 wird die Architektur der Teilsysteme beschrieben, die die Prüfungsfunktionalität bilden.

```
context Activity
inv invConformsSucceedence =
  not self.typedAs.oclIsUndefined() implies
  self.typedAs.adjRCEs(RelConKind::SUCCEEDENCE)->forAll(r |
    r.matchUpperBound(self.countMaxTypeSucc(r.target)) and
    r.matchLowerBound(self.countMinTypeSucc(r.target)))
```

Listing 4.7: OCL-Operationen und -Invariante zur Auswertung von succeeded_by-Bedingungsbeziehungen

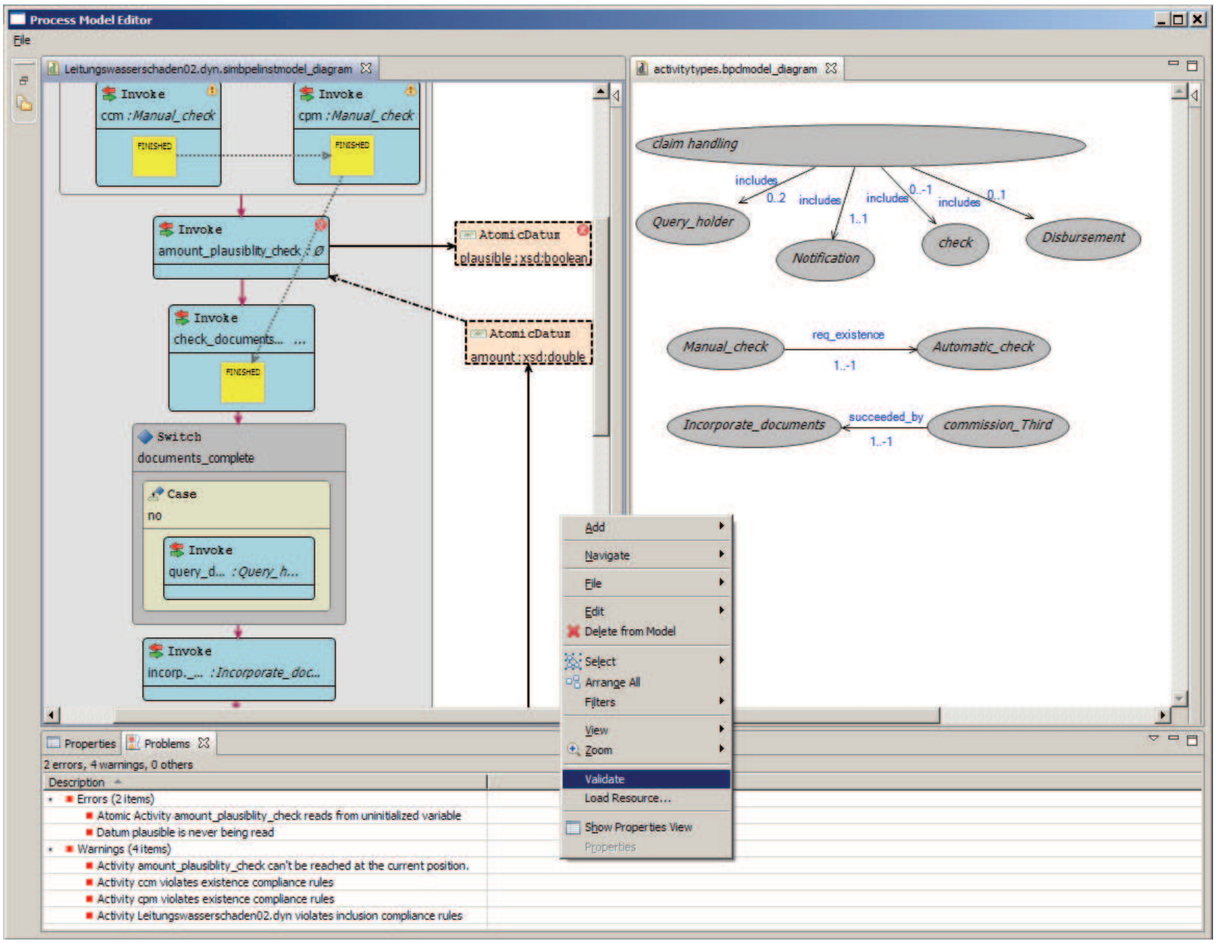


Abbildung 4.10: Inkorrektes und inkompliantes Prozessinstanzmodell im Prozessmodelleditor

4.5.1 Prozessmodelleditor

Abbildung 4.10 zeigt einen Screenshot des Prozessmodelleditors, nachdem eine Prüfung des links geöffneten Prozessinstanzmodells durchgeführt wurde. Der gezeigte Ausschnitt ist dem Prozessinstanzmodell aus Abbildung 1.12 entnommen. Rechts in der Abbildung ist das Prozesswissensmodell dargestellt, das fachliche Bedingungen spezifiziert, die vom Prozessinstanzmodell nicht vollständig eingehalten werden. Unten ist eine Problemliste mit gefundenen Verletzungen dargestellt, gruppiert in Fehler (Errors) und Warnungen (Warnings). Die Prüfung eines Prozessinstanzmodells wird durch Auslösen des Kommandos Validate im Kontextmenü des Prozessinstanzmodells gestartet.

Das Prozessinstanzmodell verletzt Bedingungen an mehreren Stellen:

write-only Prozessvariable Die Prozessvariable mit Namen plausible wird nur geschrieben, jedoch nie gelesen. Dies ist eine Verletzung einer Korrektheitsbedingung. Zwar führt diese Art von Inkorrektheit nicht zu technischen Problemen, ist aber unabhängig von einer bestimmten Domäne nicht sinnvoll. Neben der Fehlermarkierung wird der Fehler in der Problemliste unter Errors aufgeführt mit der Beschreibung “Datum plausible is never being read”, die den Prozessbeteiligten über die Art und Ursache der Verletzung aufklärt.

uninitialisierte Prozessvariable Die Prozessvariable amount wird vor ihrem ersten Lesezugriff nicht geschrieben (sondern erst später im Prozess). Diese Art von Inkorrektheit wurde in Beispiel 4.2 erläutert. Da die betreffende OCL-Invariante für die lesende Aktivität definiert ist, wird im Prozessinstanzmodell die Aktivität *amount_plausibility_check* mit einem Fehlermarker versehen und eine Fehlermeldung mit dem Wortlaut “Atomic Activity amount_plausibility_check reads from uninitialized variable” in der Problemliste aufgeführt.

keine automatischen Checks Im Prozessinstanzmodell finden sich Aktivitäten zur manuellen Prüfung der materiellen bzw. formellen Deckung. Diese werden nicht durch automatische Checks begleitet, wie das Prozesswissensmodell vorschreibt. Daher werden beide Aktivitäten mit Warnungsmarkierungen versehen. Der Hinweis in der Problemliste lautet “Activity ccm violates existence compliance rules”; ccm ist dabei die Abkürzung für “check coverage manually”. Für cpm (check payment manually) existiert ein analoger Eintrag.

überzählige und fehlende Aktivitäten Im Prozessinstanzmodell gibt es zwei Aktivitäten `dc1` und `dc2` vom Typ *disbursement*, die später im Prozess ausgeführt werden und im Screenshot dargestellten Ausschnitt des Prozessinstanzmodells nicht sichtbar sind. Dieser Sachverhalt ist inkompliant zur Forderung des Prozesswissensmodells, nach der nur maximal eine *disbursement*-Aktivität im Prozess durchgeführt werden darf. Außerdem fehlt im Prozessinstanzmodell eine *Notification*-Aktivität. Beide Verletzungen verstoßen gegen includes-Bedingungsbeziehungen des Prozesswissensmodells. Diese sind als OCL-Invarianten des gesamten Prozessinstanzmodells definiert, d.h. mit Bezug zur Metaklasse `SimBpelModel`. Diese Metaklasse findet sich in der konkreten Syntax nicht direkt als visuelles Objekt wieder. Daher fehlen für diese Art von Verletzungen Markierungen im Diagramm des Prozessinstanzmodells. Jedoch wird die Verletzung in der Problemliste als “Activity Leitungswasserschaden02.dyn violates inclusion compliance rules” aufgeführt.

4.5.2 Architektur

Der Prozessmodelleditor wurde gänzlich mit generativen Werkzeugen zur Softwareentwicklung erzeugt. Der Anteil handgeschriebenen Codes beschränkt

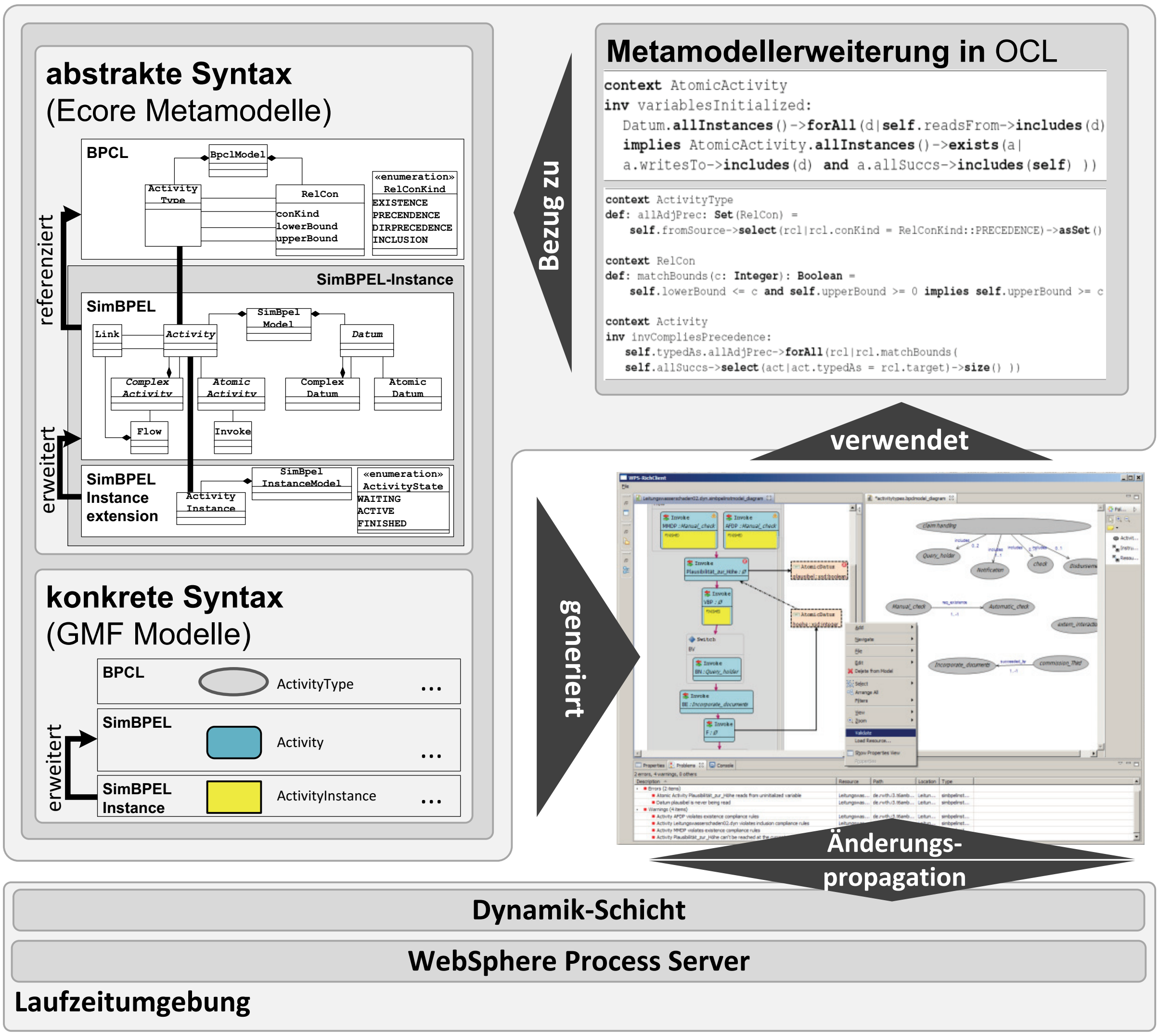


Abbildung 4.11: Grobarchitektur des Prozessmodelleditors (nach [WH08])

sich auf wenige Klassen.

In Abbildung 4.11 ist die Grobarchitektur des Prozessmodelleditors dargestellt. Die abstrakte Syntax der einzelnen Modelle ist jeweils als Metamodell in Ecore spezifiziert. Diese Metamodelle sind miteinander verschränkt, um einerseits Komplianzprüfungen realisieren zu können (Verschränkung SimBPEL mit BPCL) und existierende Festlegungen bei ähnlichen Modellierungssprachen wiederverwenden zu können (SimBPEL-Instance als Erweiterung von SimBPEL). Aus diesen Metamodellen können mittels eines Generators aus dem Eclipse Modeling Framework [BSMP09] Java-Klassen generiert werden.

Die instanziierten Laufzeitobjekte dieser Klassen speichern die reinen Modellinformationen; die konkrete Darstellung, also die konkrete Syntax wird in zusätzlichen Modellen spezifiziert, die dem Graphical Modeling Framework [Gro09, Kapitel 4] zugeordnet sind. Hier kann die Wiederverwendung der

Spezifikationen auf gleiche Weise erfolgen wie bei der abstrakten Syntax: Die Festlegungen der konkreten Syntax für SimBPEL-Instance-Modelle (Prozessinstanzmodelle) ist eine reine Erweiterung der Definition der konkreten Syntax für SimBPEL-Modelle (Prozessdefinitionsmodelle). In der Spezifikation der konkreten Syntax für SimBPEL-Instance-Modelle kommen nur Festlegungen für Aktivitätsinstanzen hinzu, die in Prozessdefinitionsmodellen fehlen. Für alle übrigen Modellelemente werden die Angaben aus der Definition für SimBPEL verwendet.

Aus den Spezifikationen zur abstrakten und konkreten Syntax kann der Prozessmodelleditor generiert werden. Das Generat ist eine Sammlung so genannter Eclipse Plug-ins [CR09], die eine rudimentäre Laufzeitumgebung um Diagrammeditoren für die verschiedenen Prozessmodellarten zum Prozessmodelleditor erweitern.

Der Prozessmodelleditor wertet beim Auslösen des Validate-Kommandos für jedes Modellelement zugehörige OCL-Invarianten aus. Die OCL-Invarianten und darin verwendete OCL-Operationen befinden sich in einem gemeinsamen Entwicklungsdokument `constraints.ocl`. Dieses Dokument wird bei der Generierung des Prozessmodelleditors nicht verarbeitet, sondern nur zur Laufzeit des Prozessmodelleditors interpretiert, kann also zur Laufzeit noch geändert werden, was die Entwicklung von OCL-Invarianten erleichtert.

Weitere Details zur Generierung und Architektur des Prozessmodelleditors sind in [WH08] nachzulesen.

4.6 Diskussion

Der gewählte Ansatz zur Prüfung der Korrektheit und Komplianz von Prozessmodellen hat gewisse Vor- und Nachteile, durch die er sich von verwandten Ansätzen (vgl. Abschnitt 4.7) abgrenzt. Diese werden im Folgenden diskutiert.

4.6.1 Vorteile des Ansatzes

Der Ansatz hat Vorteile für Endbenutzer des Prozessmodelleditors aber für Entwickler für zukünftige Erweiterungen.

Vorteile für Endbenutzer

Die Prüfungen beziehen sich einerseits auf technische Korrektheitsaspekte in den Prozessmodellen, die unabhängig von bestimmten Festlegungen eines Anwendungsbereichs wie Versicherungen sind, in dem die Prozesse durchgeführt werden. Andererseits unterstützen die Prüfungen auch die Komplianz von Prozessinstanzmodellen in Bezug auf explizit modelliertes Prozesswissen. Sowohl die Korrektheit als auch die Komplianz von Prozessmodellen ist wesentlich für eine ordnungsgemäße Durchführung dynamischer Prozesse.

Gefundene Verletzungen von Korrektheits- und Komplianzbedingungen teilt der Prozessmodelleditor unter Angabe von Fehlerposition und -art dem Benutzer mit. Diese Art der Fehlermeldung ist nützlicher als die reine Benachrichtigung, dass eine Verletzung existiert. Die Fehlerposition ist dabei das Modellelement, für das eine Invariante nicht erfüllt ist. Die Verletzung wird einerseits in der Problemliste aufgeführt, andererseits als Marker auf dem betreffenden Modellelement. In manchen Fällen kann ein Fehler nicht genau einem Element zugeordnet werden. Für denselben Fehler ist dann eine Invariante für verschiedene Elemente (derselben Metaklasse) nicht erfüllt. Beispielsweise bei einer inkorrekten, weil zyklischen Kontrollflussdefinition verletzt jede Aktivität im Zykel die zugehörige Invariante, weswegen auch jede Aktivität mit einem Marker versehen wird. Ebenso kann eine Aktivität mehrere Invarianten verletzen und erhält in diesem Fall trotzdem nur einen Marker, dafür aber pro Verletzung einen Eintrag in der Problemliste. Es wäre ebenso denkbar, die Invarianten für Komplianzbedingungen so umzuformulieren, dass sie an Aktivitätstypen oder Bedingungsbeziehungen in BPCL-Prozesswissensmodellen ausgewertet werden. Die Marker würden dann ggf. Aktivitätstypen oder Bedingungsbeziehungen in BPCL-Prozesswissensmodellen markieren. Dieser Weg wurde insbesondere aus dem Grund nicht gegangen, weil durch Spezialisierungen implizite Bedingungsbeziehungen eingeführt werden, die keine direkte Repräsentation im BPCL-Prozesswissensmodell besitzen, der eine Fehlermarkierung zugeordnet werden kann.

Änderungen in Prozesswissensmodellen wirken sich unmittelbar auf die Komplianzprüfungen aus. Da Prozesswissensmodelle lokale Modelle sind, werden Änderungen zunächst nur lokal gültig. Die Unterstützung simultaner Änderungen in Prozesswissensmodellen ist kein Gegenstand dieser Arbeit. Allerdings lässt sich dies einfach realisieren, da die Prozesswissensmodelle in einfachen Dateien gespeichert werden und dank der eingesetzten Basistechnologie Eclipse eine einfache Versionsverwaltung beispielsweise über das Versionsverwaltungssystem Subversion [CSFP04] möglich ist.

Vorteile aus Entwicklersicht

Die (Weiter-)Entwicklung des Prozessmodelleditors geschieht modellgetrieben: Der handgeschriebene Code beschränkt sich auf eine Klasse zur Initialisierung des OCL-Interpreters. Sämtlicher Code, der die diagrammtische Darstellung und Funktionen zur Manipulation der Prozessmodelle implementiert, kann automatisch aus den Metamodellen generiert werden. Da wenig handgeschriebener Code erforderlich ist, ist der Ansatz mit wenig Aufwand auf andere Prozessmodellierungssprachen übertragbar. Die Modellierung von Prozessdefinitions- und -instanzmodellen in beispielsweise BPMN [Obj08] ist auf gleiche Weise machbar. Der Fokus dieser Arbeit liegt auf der Sprache WS-BPEL, da sie die Sprache der Laufzeitumgebung WebSphere Process Server ist. Anzupassen wären hierbei das Metamodell für Prozessdefinitionsmodelle und die OCL-Ausdrücke, die auf dieses Metamodell Bezug nehmen. BPCL für Prozesswissensmodelle kann bestehen bleiben, da sich in BPCL keine Spezifika wiederfinden, die nur für WS-BPEL gelten.

Die Einheitlichkeit des Ansatzes zeigt sich am offensichtlichsten in der einheitlichen Metamodellierung der Prozessmodellierungssprachen. Hierzu war eine Übertragung der Sprachdefinition von WS-BPEL als XML Schema in ein Metamodell nötig, das in Ecore verfasst ist. Diese Übertragung lässt einige Sprachdetails aus, weswegen die Sprache nach der Übertragung zur Unterscheidung SimBPEL genannt wird. Durch die einheitliche Metamodellierung in Ecore ist die in Unterabschnitt 4.4.1 beschriebene Verschränkung der Metamodelle erst möglich.

Die Verschränkung ist Voraussetzung für die einheitliche Realisierung von Korrektheits- und Komplianzprüfungen mittels OCL. Die OCL erlaubt, sowohl Korrektheits- und Komplianzbedingung in kompakter Weise als syntaktische Bedingungen an die Modelle zu formulieren. Hierdurch wird die Erweiterbarkeit der Menge aller Korrektheits- und Komplianzbedingungsarten gefördert.

Die einheitliche Metamodellierung der Prozessmodelle ermöglicht die Wiederverwendung syntaktischer Festlegungen. So ist die Syntax für Prozessinstanzmodelle in SimBPEL-Instance von der für Prozessdefinitionsmodelle in SimBPEL abgeleitet. Syntaktisch sind SimBPEL-Instance-Modelle im Wesentlichen SimBPEL-Modelle zuzüglich Ausführungsinformationen. Das Metamodell für SimBPEL-Instance ist daher lediglich eine Erweiterung des SimBPEL-Metamodells.

Da die Korrektheits- und Komplianzprüfungen allein auf der Syntax der Prozessmodelle durchgeführt werden, können alle Aspekte eines Prozesses berücksichtigt werden, die sich in der Syntax der Prozessmodelle wiederfinden. Da beispielsweise der Datenfluss in SimBPEL-Modellen und somit auch in SimBPEL-Instance-Modellen explizit modelliert wird, ist eine Prüfung wie in Beispiel 4.2 möglich, die sowohl die Kontrollfluss- als auch die Datenflussdefinition miteinbezieht.

Der Einheitlichkeit ist zu Verdanken, dass die Prüfungen nicht nur mehrere Aspekte abdecken, sondern auch auf alle Modellierungsebenen angewendet werden können. Die Korrektheitsbedingungen aus Abschnitt 4.3 sind sowohl auf Prozessdefinitionsmodelle in SimBPEL als auch Prozessinstanzmodelle in SimBPEL-Instance anwendbar. Spezielle OCL-Korrektheitsbedingungen können formuliert werden, um die Korrektheit von Prozesswissensmodellen zu garantieren.

4.6.2 Nachteile des Ansatzes

Das in diesem Kapitel beschriebene Vorgehen hat einige, teils ansatzbedingte Schwächen, die im Weiteren beschrieben werden.

Grenzen der Ausdrucksmächtigkeit von OCL

Die Korrektheits- und Komplianzprüfungen werden allein anhand der Modellsyntax durchgeführt. Die Modelle sind Prozessmodelle, beschreiben also ein Verhalten, nicht etwa eine statische Struktur.

Die OCL-Operationen aus Unterabschnitt 4.3.1 basieren stark auf der Kompositionsstruktur der Prozessinstanzmodelle, also der Hierarchie, die durch komplexe Aktivitäten und darin enthaltene (komplexe oder atomare) Aktivitäten gebildet wird. Zudem vereinfacht die Korrektheitsforderung der WS-BPEL-Spezifikation, dass durch Links gebildete Kontrollflüsse azyklisch sein müssen, die Berechnung der Kontrollflussabhängigkeit zweier Aktivitäten [AAA+07, Unterabschnitt 11.6.2].

Grundsätzlich können für einfache Prozessmodellierungssprachen wie K3 [EKL+08, Unterabschnitt 2.4.4] – eine Abwandlung von UML-Aktivitätsdiagrammen zur Modellierung verfahrenstechnischer Arbeitsprozesse – mit dem gleichen syntaktischen Ansatz automatische Prüfungen realisiert werden, auch wenn K3 zyklische Kontrollflüsse erlaubt [HW09]. Der Ansatz stößt bei Prozessmodellierungssprachen wie WF-Nets [Aal97, Definition 6], die komplexere Kontrollflussdefinitionen zulassen [AtHKB03], bzgl. bestimmter Korrektheitsforderungen wie Verklemmungsfreiheit insoweit an seine Grenzen, dass hier so viele Spezialfälle abgehandelt werden müssten, dass andere Techniken erforderlich wären [Ver04].

Typisierung

Eine Schwäche des Ansatzes ist, dass für Komplianzprüfungen der Bezug zwischen Prozessinstanz- und Prozesswissensmodellen über Typisierung der Aktivitäten erst hergestellt werden muss. Untypisierte Aktivitäten haben keinen Bezug zu Aktivitätstypen. Komplianzverletzungen, die diese Aktivitäten betreffen, können nicht festgestellt werden. Umgekehrt können untypisierte Aktivitäten nicht zur Komplianzeinhaltung oder -verletzung anderer Aktivitäten beitragen. Hierdurch werden Komplianzprüfungen verfälscht.

Das Problem ist nicht ansatzbedingt. Vielmehr ist es ein prinzipielles Problem, dass aus der Modellinformation einer Aktivität nicht ihre fachliche Bedeutung hervorgeht. In verwandten Ansätzen wird davon ausgegangen, dass die Aktivitäten stets so benannt werden, dass über reine Namensvergleiche der Bezug zum modellierten Prozesswissen hergestellt werden kann. Diese Annahme konnte in dieser Arbeit nicht getroffen werden, da fachlich gleichbedeutende Aktivitäten beliebig unterschiedlich benannt werden können: Beispielsweise sind mit “Ermittlung Exkassoempfänger” und “Zahlungsempfänger feststellen” benannte Aktivitäten fachlich identisch, der namentliche Vergleich ergibt jedoch offensichtlich keinerlei Ähnlichkeiten.

Qualität und Vollständigkeit der Prozesswissensmodelle

In Prozesswissensmodellen wird Prozesswissen explizit festgehalten. Die Pflege von Prozesswissensmodellen verursacht allerdings zusätzlichen Aufwand. Für die Unterstützung von Prozessen sind sie nicht unmittelbar notwendig. Daher besteht die Gefahr, dass Prozesswissensmodelle nicht ausreichend gepflegt und daher unvollständig sind, also nicht alle fachlichen Bedingungen an Prozesse abdecken.

Des Weiteren können Prozesswissensmodelle falsches oder veraltetes Prozesswissen widerspiegeln. Zwar lassen sich Prozesswissensmodelle auf Korrektheit überprüfen, wie in Unterabschnitt 4.3.2 gezeigt. Ob das ausgedrückte Wissen allerdings selbst fachlich sinnvoll und aktuell ist, lässt sich mit informatischen Mitteln nicht prüfen.

Im nächsten Kapitel ist beschrieben, wie sich die Probleme mildern lassen, die mit der Nutzung expliziten Wissens im Prozesswissensmodellen einhergehen. Hierbei wird statt des expliziten Wissens der Prozesswissensmodelle das implizite Prozesswissen genutzt, dass in Prozessdefinitions- und -instanzmodellen vorhanden ist.

4.7 Verwandte Arbeiten

In diesem Abschnitt wird unser Ansatz zur Korrektheits- und Komplianzprüfung mit ähnlichen Ansätzen aus der Forschung verglichen. Eine exakte Kategorisierung aller Eigenschaften von Ansätzen in diesem Bereich ist nicht möglich, da bestimmte Eigenschaften sich teils gegenseitig bedingen oder ausschließen. Zwei Merkmale lassen sich jedoch bei jedem Ansatz feststellen: Erstens zielt jeder Ansatz entweder auf die Einhaltung der technischen Korrektheit (vgl. Unterabschnitt 4.7.1) oder auf die fachliche Komplianz (vgl. Unterabschnitt 4.7.2) von Prozessmodellen ab, jedoch keiner auf beides. Zweitens unterscheidet sich die algorithmische Umsetzung der Korrektheits- bzw. Komplianzprüfungen jeweils darin, ob sie auf der Prozessmodellsyntax oder

-semantik operiert, wobei letztere i.A. ein Transitionssystem bzw. ein Markierungsgraph für Petri-Netze ist.

4.7.1 Korrektheitsprüfungen

Syntaxbasierte Ansätze

Graphreduktion In [DAV05, MA07] beschreiben Aalst et al. – basierend auf einer Idee von Sadiq et al. [SO00] – einen Ansatz, anhand dessen komplizierte Korrektheitsbedingungen in EPC-Prozessdefinitionsmodellen [NR02] überprüft werden können.

Ansatz Basierend auf einer Petri-Netz-Semantik für EPC-Prozessdefinitionsmodelle und einer auf Petri-Netz-Markierungen definierten Fehlerfreiheitsbedingung ("soundness") [Aal97] definieren Aalst et al. in informeller Weise eine Graphgrammatik mit Regeln $\mathcal{R}_{red}$, die auf ein fehlerfreies bzw. fehlerhaftes EPC-Prozessdefinitionsmodell angewendet werden können und in jedem Transformationsschritt wiederum ein fehlerfreies bzw. fehlerhaftes EPC-Modell erzeugen. Durch sukzessive Anwendungen von Regeln $\mathcal{R}_{red}$ kann aus einem EPC-Modell ein bzgl. $\mathcal{R}_{red}$ syntaktisch minimales EPC-Modell konstruiert werden. Dessen semantikdefinierendes Petri-Netz kann anschließend als Eingabe anderweitiger, semantikbasierter Verifikationsverfahren verwendet werden.

Diskussion Der Ansatz von Aalst et al. unterscheidet sich stark von unserem, beginnend mit den *unterschiedlichen Prozessmodellierungssprachen* EPC und WS-BPEL. Graphersetzungen sind sicherlich ein mächtiges Mittel zur syntaktischen Analyse von Modellen graphischer Sprachen. Offen bleibt jedoch, inwieweit die Graphersetzungen tatsächlich *formalisiert* wurden. Die Erläuterungen hierzu in [DAV05, MA07, Abschnitte 4] lassen den Schluss zu, dass es sich bei den Graphersetzungregeln eher um Veranschaulichungen handelt, die anderweitig implementiert wurden, da die gezeigten Regeln offensichtliche Mehrdeutigkeiten enthalten. Bei den Graphreduktionen handelt es sich zudem nur um vorbereitende Maßnahmen zur syntaktischen Verkleinerung von EPC, die in syntaktisch ursprünglicher Form für Verifikationen aufgrund zu großer Transitionssysteme (Markierungsgraphen) ungeeignet sind. Vorteil dieses zweistufigen Ansatzes ist, dass durch eine auf der Prozessmodellsemantik basierende Verifikation auch subtile Inkorrektheiten ermittelt werden, z.B. bestimmte Verklemmungen, die durch rein syntaktische Analysen nicht gefunden werden können. Dies setzt allerdings voraus, dass das zu analysierende EPC-Modell hinreichend klein ist bzw. sich mit den besagten Reduktionen hinreichend verkleinern lässt, was nicht garantiert werden kann. Ein generelles Problem handeln sich die Autoren spätestens durch die abschließende Verifikationsphase ein: Das Verifikationsergebnis ist lediglich dreiwertig (korrekt, bedingt korrekt und inkorrekt) und hat keinerlei Bezug mehr zum ursprünglichen EPC-Modell und ist daher für Prozessbeteiligte von begrenztem Nutzen. Die Abbildung 7 in [MA07] suggeriert zwar, dass sich die Ergebnisse zu detaillierten Annotationen mit Bezug zum ursprünglichen EPC-Modell erweitern lassen, wie dies geschieht, ist allerdings nicht veröffentlicht.

Syntaxsteuerung in Aristaflow Im ADEPT-Prototyp [RD98, Rei00] und im Nachfolgesystem Aristaflow [DR09] können Prozessinstanzmodelle nur strikt syntaxgesteuert geändert werden.

Ansatz Durch die Syntaxsteuerung wird erreicht, dass die Manipulation eines korrekten Prozessinstanzmodells stets wiederum ein korrektes Prozessinstanzmodell erzeugen muss. Die Autoren nennen dies “correctness by construction”. Der Kern des Ansatzes sind Regeln eines Graphkalküls, die korrektheitsbewahrend sind. Inkorrekte Zustände nach Anwendung einer Regel sind nicht möglich, inkorrekte Zwischenzustände können nicht erzeugt werden, da die Regeln nur atomar anwendbar sind.

Diskussion Aufgrund der gleichen Ausrichtung auf dynamische Prozesse sind nicht nur Prozessdefinitionsmodelle sondern insbesondere auch Prozessinstanzmodelle mögliche Objekte von Manipulationen. Offensichtlich fassen die Autoren den Unterschied zwischen Prozessdefinitionsmodell und Prozessinstanzmodell auf wie in dieser Arbeit beschrieben, nämlich dass Prozessdefinitionsmodelle Prozessinstanzmodelle sind, die keine Zustandsinformationen besitzen. Syntaxsteuerung zur Korrektheitsbewahrung ist reizvoll, da ungewollte Situationen per se ausgeschlossen sind. Ihr wesentlicher Nachteil ist jedoch, dass die Wirkungsweise von Kalkülregeln gegenüber kleinschrittigen Änderungen (Aktivitäten erzeugen, Kontrollflüsse zwischen zwei Aktivitäten ziehen etc.) schwieriger abzusehen ist. Dies mag von Prozessbeteiligten als Gängelung aufgefasst werden. Syntaxgesteuerte Editoren haben sich bereits in anderen Gebieten, beispielsweise Softwarenentwicklungsumgebungen, aus diesem Grund nie durchsetzen können.

Semantikbasierte Ansätze

In semantikbasierten Ansätzen ist die Semantik eines Prozessmodells Gegenstand der Analyse. Diese Ansätze erfordern natürlich, dass die Semantik, also ein Objekt der semantischen Domäne, in irgendeiner Form vorliegt, d.h.

erzeugt werden muss. Hierfür werden Mittel wie Model Checker eingesetzt, die ihre Ursprünge in der Verifikation komplexer Hardwaresysteme haben.

Korrektheit von WS-BPEL-Modellen (Bianculli et al.) Bianculli et al. beschreiben in [BGS07] einen Ansatz zur Überprüfung gewisser Korrektheitseigenschaften von einander aufrufenden WS-BPEL-Prozessdefinitionsmodellen.

Zweck Dadurch, dass generell ein Verbund von Prozessdefinitionsmodellen einbezogen wird, ist es möglich, Eigenschaften einer Prozessmodellierung zu prüfen, deren Modelle aufgrund beispielsweise organisatorischer Überlegungen auf mehrere, miteinander kommunizierende Laufzeitumgebungen verteilt sind. Als geprüfte Eigenschaften sind in [BGS07] beispielhaft Wertebereiche von Prozessvariablen aufgeführt, die bei der Prozessdurchführung nicht verlassen werden dürfen.

Ansatz Das Vorgehen des Ansatzes beruht im Wesentlichen auf einer Transformation von WS-BPEL in die Eingabesprache des Model Checkers Bogor [DBL03]. Zu prüfende Eigenschaften können dabei direkt als Annotationen in den WS-BPEL-Prozessdefinitionsmodellen verfasst werden.

Diskussion Die Arbeit von Bianculli et al. fokussiert in erster Linie Korrektheitsbedingungen. Wie sich fachliche Bedingungen in WS-BPEL modellieren und prüfen lassen, geht aus den Veröffentlichungen nicht hervor. Falls eine Bedingung verletzt wird, gibt Bogor ein Gegenbeispiel aus, das eine Prozessdurchführung darstellt, in der die betreffende Bedingung verletzt wird. Wie genau das Ergebnis präsentiert wird, geht aus den Veröffentlichungen ebenfalls nicht hervor, insbesondere nicht, wie der Bezug zur Fehlerquelle der Eingabe, also des WS-BPEL-Prozessdefinitionsmodells, hergestellt werden kann.

Korrektheit von Petri-Netzen (Verbeek et al.) Verbeek et al. befassen sich in [VA00, VBA01, Ver04] mit der Verifikation von Petri-Netzen.

Zweck Petri-Netze bilden, wie in Abschnitt 2.4 erläutert, eine geeignete formale Basis bzw. semantische Domäne für Prozessmodellierungssprachen, beispielsweise auch für WS-BPEL [OVA^{+}07]. Dementsprechend nützlich sind Analyseverfahren für Petri-Netze.

Ansatz Verbeek et al. machen Gebrauch von existierenden Analysetechniken für Petri-Netze und insbesondere solchen, die auf Markierungsgraphen von Petri-Netzen basieren. Voraussetzung hierfür ist, dass das jeweilige Prozessdefinitionsmodell zuvor in ein Petri-Netz übersetzt wird, das Eingabe des prototypischen Analysewerkzeugs WofBPEL ist.

Diskussion WofBPEL findet subtile Inkorrektheiten in WS-BPEL-Modellen, die von Korrektheitsprüfungen der vorliegenden Arbeit nicht erfasst werden. Dazu gehört beispielsweise die Erkennung von parallel ausgeführten Aktivitäten, die um eintreffende, externe Nachrichten konkurrieren. Erkauft wird diese Genauigkeit der Analyse allerdings dadurch, dass für bestimmte Analysen die Erzeugung und Traversierung von Markierungsgraphen erforderlich ist, was vergleichsweise viel Speicherplatz und Laufzeit in Anspruch nimmt. Ein für diese Ansätze typisches Problem ist wiederum die Rückführung der Ergebnisse in der Art, dass Fehler sich auf die syntaktische Struktur des Prozessmodells beziehen. Gefundene Fehler werden auch hier in Form von möglichen, aber fehlerhaften Prozessläufen dargestellt [VBA01, Abbildungen 19 und 22]. Darüberhinaus ist die Abbildung einer Prozessmodellierungssprache in eine semantische Domäne wie Petri-Netze nicht trivial und dementsprechend fehleranfällig gegenüber einer rein syntaktischen Erfassung in Form eines Metamodells.

4.7.2 Komplianzprüfungen

Allgemeine Anforderungen (Ly et al.) Anforderungen für eine werkzeuggestützte Prüfung von Prozesswissen sind in [LRD06, LGRMD08, LRD08] beschrieben. Ly et al. erkennen ebenfalls, dass reine Korrektheitsprüfungen in Prozessmodellen nur einen Teil aller ungewollten Situationen aufspüren können. In ihrer Anwendungsdomäne, dem Klinikbetrieb, sind beispielsweise die Verabreichung inkompatibler Medikamente sowie bestimmte Aktivitätsreihenfolgen, wie die Verabreichung von Blutverdünnungsmitteln wie Aspirin vor chirurgischen Eingriffen aus naheliegenden, fachlichen Erwägungen zu vermeiden.

Ansatz Ly et al. schlagen eine Formalisierung von Prozesswissen auf Basis von Quintupeln der Form $(type, source, target, position, userDefined)$ vor. Ein Quintupel besagt, (1) um welchen Typ ($type$) von Bedingung es sich jeweils handelt, beispielsweise gegenseitiger Ausschluss oder Abhängigkeit von Aktivitäten, (2) auf welche zwei Aktivitäten $source$ und $target$ sich die Bedingung bezieht, (3) ob eine bestimmte Reihenfolge ($position$) zwischen den Aktivität eingehalten werden muss und (4) von welcher Wichtigkeit die jeweilige Regel ist ($userDefined$). Prozessinstanzmodelle können dann gegen eine Menge derartiger Regeln geprüft werden.

Diskussion Der Ansatz von Ly et al. zeigt deutliche Parallelen zu unserem. So ähneln die Quintupel offensichtlich den Bedingungsbeziehungen aus Unterabschnitt 2.2.3. Unterschiedlich sind beide Formen der Prozesswissensformalisierung jedoch beispielsweise darin, dass Inklusions-Bedingungsbeziehungen in den Quintupeln nicht ausgedrückt werden können und Multiplizitäten nicht darstellbar sind. Zudem ist die Syntax, in der die Quintupel letztlich werkzeugseitig ausgedrückt werden sollen, nicht formalisiert. Eine weitere Übereinstimmung ist, dass generell Prozessinstanzmodelle und nicht nur Prozessdefinitionsmodelle geprüft werden.

Semantikbasierte Ansätze

Ausführbare Prozesswissensmodelle (Pesic et al.) DECLARE von Pesic et al. [PSA07, Pes08] ist eine Methode zur Entwicklung deklarativer Prozessmodellierungssprachen und eine dazu passende Ausführungsunterstützung. Konkrete DECLARE-basierte Sprachen sind ConDec [PA06] zur Unterstützung manuell durchgeführter Prozesse und DecSerFlow [AP06] zur Koordination von WebServices. Die Prozessmodellierungssprachen ähneln dabei den Prozesswissensmodellen aus Unterabschnitt 2.2.3.

Zweck Pesic et al. bezwecken die Unterstützung von hochdynamischen Geschäftsprozessen durch Ausführung von Prozessdefinitionsmodellen in DECLARE-basierten Sprachen, im Folgenden verkürzt DECLARE-Modelle genannt. Für teils statische Geschäftsprozesse wurde eine Integration mit der imperativen, Petri-Netz-basierten Sprache YAWL [AH05] in der Form realisiert, dass YAWL- und DECLARE-Modelle einander aufrufen können.

Ansatz Ein DECLARE-Modell ist eine Menge von Aktivitäten, wobei Aktivitäten im DECLARE-Sinn Aktivitätstypen unseres Ansatzes entsprechen, sowie eine Menge von LTL-Formeln (vgl. Unterabschnitt 2.6.2). Eine LTL-Formel schränkt mögliche Ablaufreihenfolgen zwischen zwei Aktivitäten ein. DECLARE-Modelle haben eine graphische Syntax, in der ein Knoten eine Aktivität und eine Kante eine LTL-Formel repräsentiert. Kanten sind getypt, wobei jedem Kantentyp eine bestimmte graphische Notation (in Form einer bestimmten Kantendekoration) zugeordnet wird. Außerdem ist mit jedem Kantentyp eine LTL-Formelvorlage assoziiert, in der die den Aktivitätsnamen entsprechenden atomaren Aussagen offengelassen sind. Zur Ausführungsunterstützung werden die LTL-Formeln in endliche Automaten übersetzt, die die Durchführung eines konkreten Prozesses insoweit steuern, dass sie dem Benutzer Rückmeldung darüber geben, welche Aktivitäten in einem bestimmten Prozesszustand gemäß des DECLARE-Modells gerade ausführbar sind.

Gemeinsamkeiten Die wesentliche Gemeinsamkeit von DECLARE-Modellen mit Prozesswissensmodellen ist, dass ablaufbezogene Konformitätsbedingungen graphisch notiert werden können. Die Semantik von Prozesswissensmodellen stützt sich ebenfalls zum Teil auf Sprachen der temporalen Logik ab. Des Weiteren können Korrektheitsverletzungen sowohl in DECLARE-Modellen, als auch in Prozesswissensmodellen automatisch gefunden werden.

Unterschiede Trotz der oberflächlichen Ähnlichkeit unseres Ansatzes mit DECLARE, existieren gewichtige Unterschiede:

Verwendung DECLARE-Modelle werden direkt zur Ausführung verwendet, Prozesswissensmodelle jedoch nur zur Komplianzprüfung ausgeführter Prozessinstanzmodelle.

Effizienz Zwar ist die Semantik von Prozesswissensmodellen mithilfe von Formeln der temporalen Logik definiert, die Überprüfung der Konformität von Prozessinstanzmodellen geschieht jedoch auf Basis von OCL. Neben Effizienzvorteilen – die generierten Automaten in DECLARE wachsen exponentiell mit der Länge der LTL-Formel [Pes08, Seite 227] – ergeben sich bereits diskutierte Vorteile in der Darstellung von Konformitätsverletzungen.

Ausdrucksstärke Jedem Kantentyp in DECLARE-Modellen ist eine LTL-Formel zugeordnet. Hierdurch wird die Ausdrucksmächtigkeit von DECLARE-Modellen begrenzt. Wo Bedingungsbeziehungen in BPCL-Prozesswissensmodellen durch Multiplizitäten attributiert werden können, was sich auf die Form der definierenden CTL-Formel auswirkt, wie in Abschnitt 2.6 erläutert, werden in DECLARE Multiplizitäten implizit und zumeist als 1..1 in den definierenden LTL-Ausdrücken für Kantentypen fest kodiert.

Nichtdeterminismus Die Flexibilität durch die deklarative Natur von DECLARE wird zwangsweise durch Nichtdeterminismus bei der Prozessdurchführung erkauft. Prozessbeteiligte müssen sich in jedem Prozesszustand im Regelfall zwischen verschiedenen, als nächstes auszuführenden Aktivitäten entscheiden. Aufgrund dieser notwendigen Interaktionen mit Prozessbeteiligten ist der Ansatz für Prozesse mit signifikantem Automatisierungsanteil unbrauchbar.

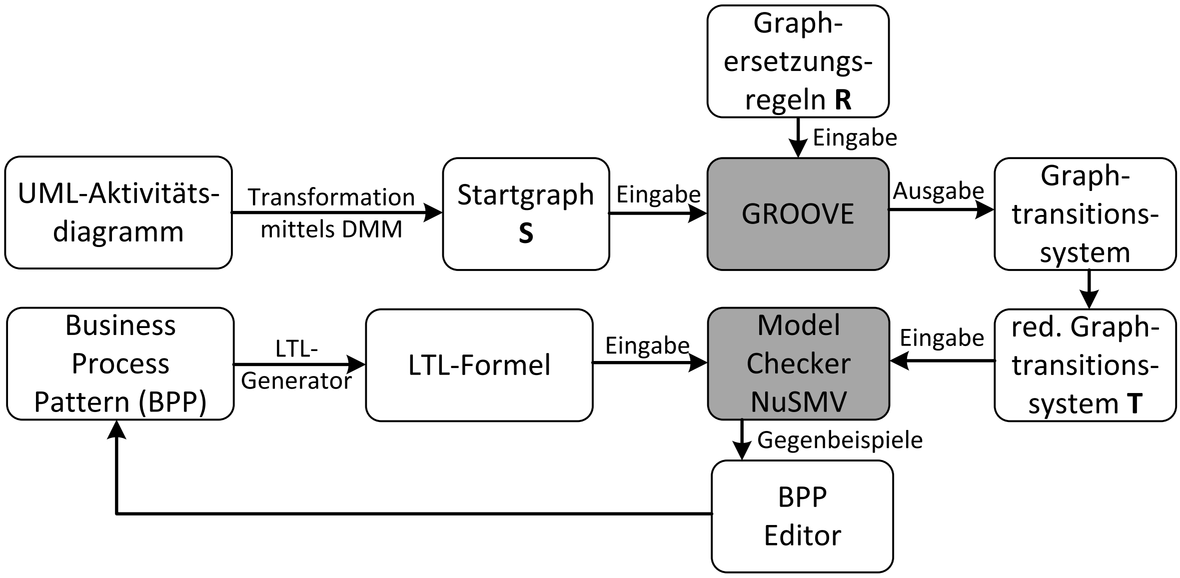


Abbildung 4.12: Ablauf zur Komplianzprüfung von Aktivitätsdiagrammen gegen Business Process Patterns

Modellierungsaspekte In DECLARE-Modellen können nur reine Ausführungspräzedenzen, also indirekte Kontrollflussdefinitionen modelliert werden, beispielsweise Datenflüsse aber nicht. Dementsprechend können Korrektheitsprüfungen, die sich auf Datenflüsse beziehen auch nicht durchgeführt werden, im Gegensatz zu unserem Ansatz.

UML-Prozesswissensmodelle (Förster et al.) Förster et al. beschreiben in [FESS07, För08], wie eine Sprachvariante von UML-Aktivitätsdiagrammen namens “Process Pattern Specification Language” (PPSL) so zur Formalisierung von Prozesswissen verwendet werden kann, dass die Komplianz von UML-Aktivitätsdiagrammen gegenüber in PPSL verfassten, so genannten Business Process Patterns (vgl. Spalte PPSL in Tabelle 4.1), werkzeuggestützt geprüft werden kann.

Ansatz Der grobgranulare Ablauf der Werkzeugunterstützung ist in Abbildung 4.12 dargestellt. Zentral ist hierbei der Model Checker NuSMV, der LTL-Formeln in Transitionssystemen evaluieren kann und im Falle der Nichterfüllung Gegenbeispiele ausgibt.

Transitionssystem Aus einem UML-Aktivitätsdiagramm wird über Transformatoren aus dem Dynamic-Meta-Modeling-Projekt (DMM) ein GROOVE-Startgraph S generiert [EHHS00, HHS04, Hau05]. Aus der Grammatik (S, R) wird ein Graphtransitionssystem T erzeugt, das anschießend so reduziert wird, dass im Ergebnis T_{red} nur noch Transitionen enthalten sind, die Finalisierungen von Aktivitäten darstellen.

LTL-Formel Eine LTL-Formel wird über einen Transformator aus einem Business Process Pattern generiert.

Diskussion Die Ähnlichkeiten bzgl. der Ausgangssituation zwischen dem Ansatz dieser Arbeit und dem von Förster et al. sind offenkundig: Prozessinstanzmodelle bzw. -definitionsmodelle sollen auf Komplianz gegen Prozesswissen geprüft werden, das in BPCL bzw. PPSL formuliert ist. Der Hauptunterschied zwischen beiden Ansätzen ist, dass der Ansatz von Förster et al. zur Werkzeugunterstützung den Weg beschreitet, der in Abschnitt 2.6 zur reinen Definition der Semantik von BPCL gewählt wurde. Demgegenüber wurde für die Implementierung der Komplianzprüfungen in unserem Fall ein syntaktischer Ansatz auf Basis von OCL gewählt (vgl. Kapitel 4). Ein weiterer Unterschied ist der Entwurf der Sprache PPSL für Prozesswissensmodelle gegenüber BPCL. Die konkrete Syntax von PPSL ähnelt viel stärker den zu prüfenden UML-Aktivitätsdiagrammen als BPCL den zu prüfenden Modellen in SimBPEL(-Instance). Der Vorteil der größeren Ähnlichkeit ist, dass ohnehin notwendige Modellierungserfahrung in UML besser weiterverwendet werden kann. Allerdings kann die syntaktische Unähnlichkeit zwischen einem (Teil eines) UML-Aktivitätsdiagramms und eines Business Process Pattern auch trügerisch sein in dem Sinne, dass das Aktivitätsdiagramm trotzdem kompliant ist; der umgekehrte Fall gilt ebenso. Die Ausdrucksmöglichkeiten zwischen BPCL und PPSL unterscheiden sich ebenfalls, was später anhand von Tabelle 4.1 diskutiert wird. Es scheint weiterhin so, als ob einzelne Business Process Patterns isoliert voneinander modelliert werden. Widersprüche im Prozesswissen können wie in dieser Arbeit über Korrektheitsprüfung nicht werkzeuggestützt festgestellt werden. Unterschiedlich ist auch die Ergebnispräsentation. Während Komplianzverletzungen in dieser Arbeit im geprüften Prozessinstanzmodell markiert werden (verletzende Situation), wird im Prozessmodelleditor von Förster et al. die verletzte Stelle im Prozesswissensmodell (Business Process Pattern) markiert. Hierbei treten Mehrfachverletzungen derselben Stelle im Prozesswissensmodell allerdings nicht zutage. Da der Weg über die semantische Domäne bei Förster et al. insbesondere auch für die Implementierung gegangen wurde, sind die Länge der erzeugten LTL-Formeln und die Größe des Transitionssystems hinsichtlich der Laufzeitkomplexität eines Model-Checker-Laufs relevant. Daher wird das von GROOVE erzeugte Transitionssystem zunächst verkleinert, so dass im Wesentlichen nur noch Finalisierungs-Transitionen sichtbar sind. Als positiver Nebeneffekt lassen sich auch die LTL-Formeln einfach halten. Insbesondere machen Aussagen über direkt nächste Zustände mittels des **X**-Operators dann Sinn. Die semantikdefinierenden CTL-Formeln aus Abschnitt 2.6 sind demgegenüber länger, weil sie sich auf ein unreduziertes Transitionssystem beziehen. Da sich die

Implementierung der Komplianzprüfungen ohnehin nicht auf die Formeln und Transitionssysteme abstützt, ist dies aber kein Nachteil.

4.7.3 Fazit

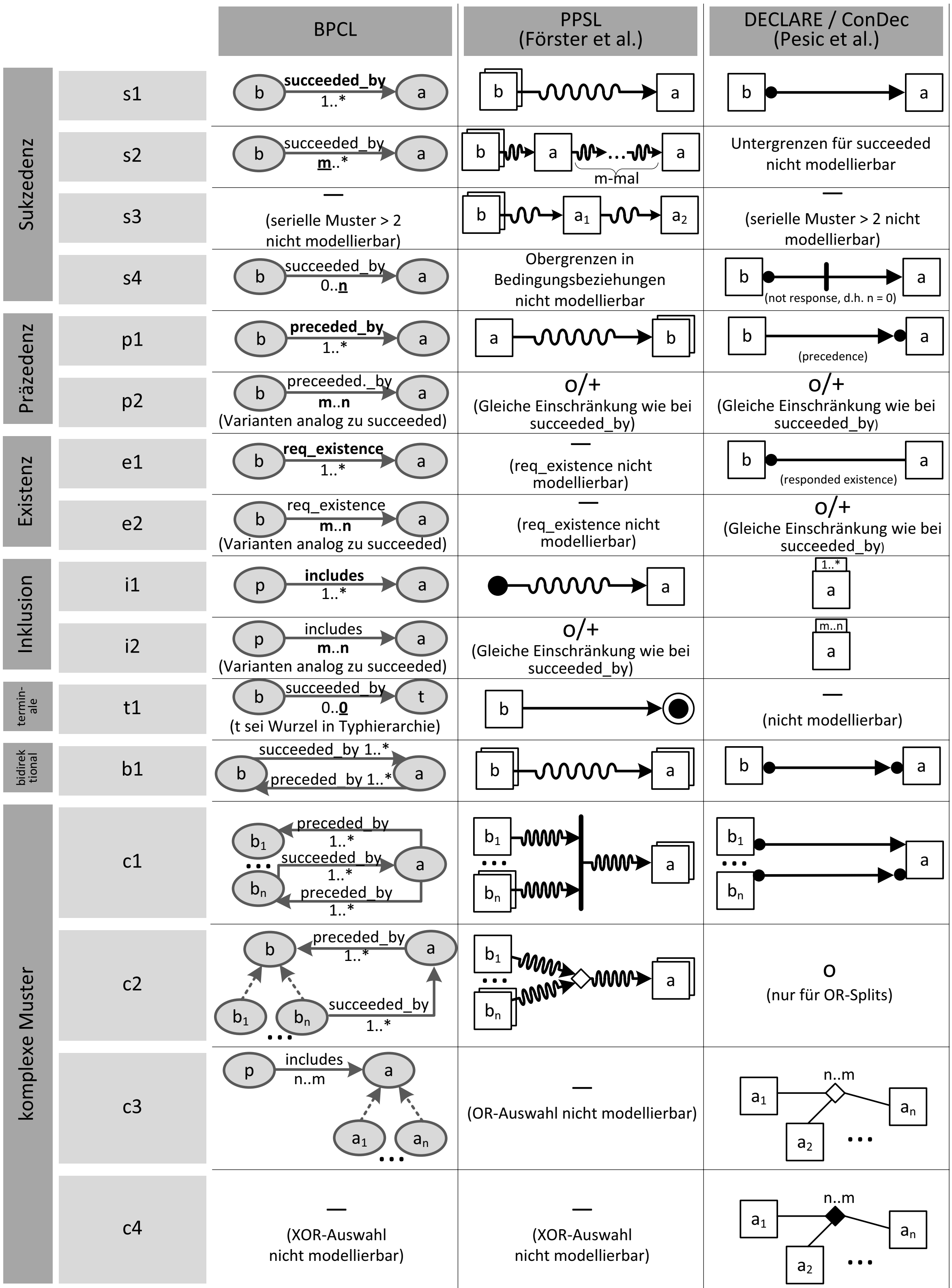
Ausdruckmöglichkeiten von Prozesswissensmodellen

BPCL aus unserem Ansatz, PPSL von Förster et al. und ConDec als DECLARE-Dialekt sind Sprachen für Prozesswissensmodelle, in denen ähnliche Aspekte auf gleichem Abstraktionsniveau diagrammatisch modelliert werden können. Es lohnt sich daher, Unterschiede in Bezug auf das Ausdrucksvermögen zwischen den Sprachen genauer zu studieren.

Tabelle 4.1 vergleicht die drei Sprachen exemplarisch. Folgende Entwurfsentscheidungen fallen dabei ins Auge:

Bedingungsteile Eine Komplianzbedingung kann in allen drei Sprachen jeweils in drei Teile zerlegt werden. Der erste Teil ist der Teil, d.h. ein oder mehrere Aktivitätstypen, für die eine Bedingung modelliert wird. Ihre Anwesenheit in Prozessinstanzmodellen, d.h. die Anwesenheit entsprechend getypter Aktivitäten, bewirkt, dass die Komplianzbedingung angewendet wird. In der konkreten Syntax von BPCL ist dies pro Bedingungsbeziehung immer der Quellaktivitätstyp oder dessen Untertypen, in der von PPSL ein, mit zwei versetzten Rechtecken notierter, «all»-Aktivitätstyp (ein Business Process Pattern drückt bei n vorhandenen «all»-Aktivitätstypen also de facto n Komplianzbedingungen aus). Der zweite Teil besteht aus Aktivitätstypen, deren Referenzierung in Prozessdefinitions- bzw. -instanzmodellen zu Erfüllung oder Verletzung der jeweiligen Komplianzbedingung beitragen kann. Bei BPCL sind das Zielaktivitätstypen, in PPSL alle Aktivitäten bis auf den «all»-Aktivitätstyp aus dem ersten Teil. In ConDec sind die Aktivitätstypen nicht ohne weiteres dem ersten oder zweiten Teil zuzuordnen. Die Zuordnung hängt hier vom dritten Teil ab. Der dritte Teil sind die Bedingungsbeziehungen zwischen Aktivitätstypen aus dem ersten und zweiten Teil. Sie werden in allen drei Fällen durch Kanten dargestellt. Die Kanten können selbst von unterschiedlichem Typ und somit unterschiedlicher Semantik sein, was durch unterschiedliche Beschriftung bzw. Dekoration der Kanten dargestellt ist. Eine Ausnahme von dieser Regel stellen Inklusionsbeziehungen in ConDEC dar (i1, i2 und c3, c4 in Tabelle 4.1). Hier ist nur der Aktivitätstyp aus dem zweiten Teil vorhanden; der inkludierende Aktivitätstyp kann als implizit modelliert angenommen werden und nicht wie im Fall von BPCL als explizit modelliert (p-Aktivität).

Ausdrucksmächtigkeit Bezüglich ihrer Ausdrucksmächtigkeit überlappen sich die drei Sprachen, jedoch inkludiert keine Sprache eine jeweils andere.



Kardinaltitäten	+ über Multiplizitäten (n..m)	o über Mehrfachmodellierung	o teils über Negation/Inkl.
Vererbung	+ explizit vorhanden	- keine Vererbung	- keine Vererbung
Abh. v. temp. Logik	+ (unabhängig)	o Ausdrucksstärke = LTL	o Ausdrucksstärke = LTL
Anz. einbezogener Akt.	- immer jeweils zwei	+ bel. viele in kompl. Mustern	+ bel. viele in kompl. Mustern
Ähnlichk. Prüfer - Prüfling	o unähnlich	o ähnlich	o unähnlich zu YAWL
Erw.bark. bzgl. Kontrollfl.	o Erw. BPCL + OCL	o PPSL-LTL-Abb. festkodiert	+ LTL-Abb. werkzeugestützt
Erw.bark. um and. Aspekt.	o Erw. BPCL + OCL	- Neukonzeptionierung	- Neukonzeptionierung

Tabelle 4.1: Ausdruckmächtigkeit verschiedener Sprachen für Prozesswissensmodelle

Alleinstellungsmerkmal von PPSL in Hinblick auf Ausdrucksvermögen sind Muster mit mehr als zwei unterschiedlichen Aktivitätstypen auf einem Pfad. Beispielsweise das Muster in Zeile s3 ist in keiner der beiden anderen Sprachen äquivalent modellierbar, die Muster in c1 und c2 sowie s2 hingegen zwar in BPCL aber nicht in DECLARE. Nur in ConDec modellierbar sind Inklusions-Bedingungsbeziehungen wie die in c4, die fordert, dass n bis m paarweise unterschiedlich getypte Aktivitäten aus der Typmenge $\{a_1, \ldots, a_n\}$ ausgeführt werden. Mitunter werden auch gleiche Bedingungen äquivalent mit unterschiedlichen Sprachmitteln modelliert. Was in PPSL durch Verzweigungen (OR-Splits) dargestellt wird, kann in BPCL durch Vererbung nachgebildet werden, wie c2 zeigt (analog c3 für OR-Inklusionsbeziehungen in ConDec).

Weitere Eigenschaften

Tabelle 4.2 stellt die in diesem Abschnitt beschriebenen verwandte Ansätze und den Ansatz dieser Arbeit einander gegenüber.

Unterstützte Prüfungsarten Ein Alleinstellungsmerkmal unseres Ansatzes ist, dass verschiedenste Prüfungen gleichsam umgesetzt wurden. Korrektheits- und Komplianzprüfungen wurde auf gleicher technischer Basis implementiert, wie in diesem Kapitel erläutert. Konsistenzbeziehungen zwischen Prozessinstanzmodellen können ebenfalls ausgewertet werden, wie in Kapitel 5 beschrieben wird. Einige verwandte Arbeiten zielen auch in diese Richtung, werden aber erst in Abschnitt 5.8 beschrieben.

Arten prüfbarer Modelle Des Weiteren sind die Prüfungsarten im Ansatz dieser Arbeit auf alle sinnvollen Ziele anwendbar: Korrektheit kann in allen drei Modellarten überprüft werden, Komplianz kann in Prozessdefinitions- und -instanzmodellen in Bezug auf Prozesswissensmodelle getestet werden. Allein die Vorläuferarbeiten von Schleicher [Sch02] im AHEAD-Projekt bieten ebenfalls Prüfungen für alle drei Modellarten, wenn hier auch aufgrund einer völlig anderen technischen und formalen Grundlage die AHEAD-Prozessmodelle gerade bei der Kontrollflussdefinition deutlich einfacher sind und die Prüfungen dementsprechend auch.

Korrektheitsprüfungen In Prozessinstanzmodellen werden in unserem Ansatz Kontrollflussdefinitionen als auch Datenflussdefinitionen beachtet. Kontrollflussdefinitionen spielen beispielsweise bei der Feststellung eine Rolle, ob eine Aktivität erreichbar ist oder nicht (vgl. Beispiel 4.3), Datenflussdefinitionen bei der Feststellung nicht-initialisierter Prozessvariablen (vgl. Beispiel 4.2). Bis auf den AHEAD- und ADEPT-Ansatz spielen bei verwandten Arbeiten nur Kontrollflussbeziehungen eine Rolle.

	Wörzberger	**AHEAD**	**ADEPT / Aristaflow** (Reichert et al.)	**PPSL** (Förster et. al)	**Graph-reduktionen** (Aalst et al.)	**Woflan** (Verbeek et al.)	**DECLARE** (Pesic et al.)
Unterstützte Prüfungsarten							
Korrektheit ➢Wissen / ➢Definition / ➢Instanz)	 ja ja ja	 teils teils teils	 nein ja ja	(s. Abs. 5.3.2) teils teils nein	 nein ja nein	 nein ja nein	 ja nein nein
Komplianz ➢Definition / ➢Instanz)	 ja ja	 ja ja	 nein nein	 ja nein	 nein nein	 nein nein	 ja ja
Konsistenz ➢Definition / ➢Instanz)	(s. Kapitel 5) ja ja	 nein nein	 nein nein	 nein nein	 nein nein	 nein nein	 nein nein
Arten prüfbarer Modelle							
Wissen	ja	ja	nein	nein	nein	nein	ja
Definition	ja	ja	ja	ja	ja	ja	ja
Instanz	ja	ja	ja	nein	nein	nein	nein
Geprüfte Aspekte bei Korrektheitsprüfungen von Prozessinstanzmodellen							
Kontrollfluss	ja	ja	ja	n/a	ja	ja	n/a
Datenfluss	ja	ja	ja	n/a	nein	nein	n/a
P.zustand	ja	ja	ja	n/a	nein	nein	n/a
Komplianzprüfungen							
Bedingungs-beziehungs-typen	Inklusion, Existenz, Präzedenz, Sukzedenz	Inklusion, dir. Präzed., dir. Sukzed.	n/a	n/a	n/a	n/a	Inklusion, Existenz, Präzendenz, Sukzedenz
Multi-plizitäten	beliebig	beliebig	n/a	teils (s. Tabelle 5.1)	n/a	n/a	nur bei Inklusion
Realisierungsansatz							
Aktivitäts-identifikation	Mapping zwischen Aktivitätstyp und Aktivität	Mapping zwischen Aktivitätstyp und Aktivität	n/a	Mapping zwischen Aktivitätstyp und Aktivität	n/a	n/a	Aktivitäts-name
Formal-isierung	semantisch (CTL*)	syntaktisch	syntaktisch (Graphkalkül)	semantisch (LTL)	syntaktisch (Graph-reduktionen)	semantisch (Markierungs-graph)	semantisch (LTL)
Implement-ierung	syntaktisch (OCL)	syntaktisch	syntaktisch	semantisch (LTL)	syntaktisch	semantisch	semantisch (LTL)
Benutzungsansatz							
Prüfungs-aktivierung ➢syntaxgest. ➢batchartig	 ja ja	nur jeweils Korrektheit / Komplianz	 nur Korrekth. ja	 nein ja	 nein ja	 nein ja	 nein ja
Ergebnis-bezug	annotiertes Instanz-modell	annotiertes Instanz-modell	annotiertes Instanz-modell	annotiertes Wissens-modell	Transitions-system	Transitions-system	n/a
Ergebnis-präsentation	detailliert, graphisch + textuell	detailliert, graphisch	detailliert, graphisch + textuell	detailliert, graphisch + textuell	unbekannt	unbekannt	textuell
Wirksamkeit Wissens-änderungen	sofort wirksam für alle Instanz-modelle	nach Neukompil-ation	n/a	vermutlich sofort wirksam	n/a	n/a	sofort wirksam

Tabelle 4.2: Korrektheits- und Komplianzprüfungsansätze

Komplianzprüfungen Hinsichtlich Komplianzprüfungen sind nur unser Ansatz, PPSL und DECLARE/ConDec zu vergleichen, wie im vorherigen Unterunterabschnitt geschehen.

Realisierungsansatz Die Realisierungen unterscheiden sich hauptsächlich darin, ob ein syntaktischer oder semantischer Ansatz gewählt wurde. In unserem Fall ist die Implementierung der Korrektheits- und Komplianzbedingungen syntaktisch mittels OCL geschehen, die Definition der Semantik von BPCL-Bedingungsbeziehungen formell auf Basis eines Graphtransitionssystems und formaler Logiken. Eine bevorzugtes Vorgehen für Korrektheitsprüfungen gibt es nicht. Teils arbeiten die verwandten Ansätze auf der Prozessmodellsyntax, teils auf der semantischen Domäne in Form eines Transitionssystems, Automaten oder Markierungsgraphen. Generell unbefriedigend gelöst ist die Abbildung zwischen Aktivitäten und Aktivitätstypen. Förster et al. typisieren Aktivitäten durch eine separate Abbildungstabelle zwischen Aktivitäten und Aktivitätstypen. Im DECLARE-Ansatz stellt sich das Problem nicht, wenn die Prozesswissensmodelle direkt ausgeführt werden. Werden sie zur Komplianzprüfung von Prozessinstanzmodellen verwendet, so scheint das Mapping über vorausgesetzte Namensgleichheit hergestellt zu werden.

Benutzungsansatz Die Werkzeugunterstützung unterscheidet sich zwischen den Ansätzen in mehreren Punkten. Die Korrektheits- und Komplianzprüfungen können in unserem Ansatz wahlweise während der Prozessmodelleditierung durchgeführt werden, wodurch eine gewisse Syntaxsteuerung erreicht werden kann. Empfehlenswert ist dies allerdings zumindest nicht für Komplianzprüfungen, da diese ggf. bewusst verletzt werden sollen. Hier empfiehlt sich eine batchartige, d.h. zu bestimmten, vom Prozessbeteiligten gewählten Zeitpunkten durchgeführte Prüfung, die ebenfalls möglich ist (vgl. Unterabschnitt 4.5.1). Im AHEAD-Prototyp werden vorhandene Korrektheitsbedingungen reaktiv während einer Editieroperation geprüft. Komplianzbedingungen können immer nur batchartig ausgelöst werden. Aristaflow und ADEPT implementieren eine reaktive Prüfungsauslösung, alle übrigen nur eine batchartige. Weitere Unterschiede ergeben sich aus der Art und Darstellung des Ergebnisses. Gerade semantische Ansätze beschränken sich oft auf die transitionssystembezogene Darstellung von Gegenbeispielen, der PPSL-Ansatz von Förster et al. bildet hier eine Ausnahme. Auch die zeitnahe Wirksamkeit von Änderungen im Prozesswissen ist wichtig: Hier macht unser Ansatz einen Fortschritt gegenüber dem Vorläufer AHEAD, bei dem bei Änderungen im Prozesswissen der Editor für Prozessinstanzmodelle stets neu übersetzt werden muss.

Kapitel 5

Implizites Prozesswissen

Prozesswissensmodelle ermöglichen die Modellierung expliziten fachlichen Prozesswissens. Wie im vorhergehenden Kapitel erläutert wurde, kann dieses Wissen für Prüfungen von Prozessinstanzmodellen verwendet werden, insbesondere nach dynamischen, also strukturellen Änderungen dieser Prozessinstanzmodelle. Ein immanenter Nachteil der expliziten Modellierung von Prozesswissen ist, dass die Erstellung und Pflege von Prozesswissensmodellen zusätzlichen Aufwand zu dem Aufwand verursacht, der durch die ohnehin notwendige Pflege von Prozessdefinitionsmodellen entsteht. Unterbleibt dieser zusätzliche Aufwand, besteht die Gefahr, dass Prozesswissensmodelle das vorhandene und veränderliche Wissen unvollständig oder fehlerhaft widerspiegeln.

Prozessdefinitionsmodelle sind für den Betrieb von Prozessmanagementsystemen unbedingt erforderlich, Prozessinstanzmodelle entstehen als von Prozessdefinitionsmodellen abgeleitete Modelle automatisch. Insofern sind viele Prozessdefinitions- und -instanzmodelle durch den Einsatz von Prozessmanagementsystemen wie dem IBM WebSphere Process Server unmittelbar vorhanden. Prozesswissensmodelle sind demgegenüber insofern optional, als dass sie für den Betrieb von Prozessmanagementsystemen nicht unbedingt erforderlich sind.

In Kapitel 4 waren Prozessinstanzmodelle stets die Prüflinge expliziter Prüfungen, Prozesswissensmodelle wurden als Prüfer verwendet. Aus den oben genannten Gründen ist es naheliegend, neben Prozesswissensmodellen auch vorhandene Prozessdefinitions- und -instanzmodelle als Kandidaten für Prüfer hinzuzuziehen.

Die Prüfungen von Prozessinstanzmodellen gegen explizites Prozesswissen in Prozesswissensmodellen werden Komplianzprüfungen genannt. Diese Prüfungen sind asymmetrisch, da bei gefundenen Widersprüchen zwischen Prozesswissens- und -instanzmodell angenommen wird, dass das Prozessinstanzmodell fachlich fehlerhaft ist. Bei der Prüfung eines Prozessinstanzmodells gegen implizites Prozesswissen, also andere Prozessinstanz- oder -definitionsmodelle, kann diese Annahme nicht getroffen werden. Daher werden Prüfungen gegen implizites Prozesswissen im Folgenden Konsistenzprü-

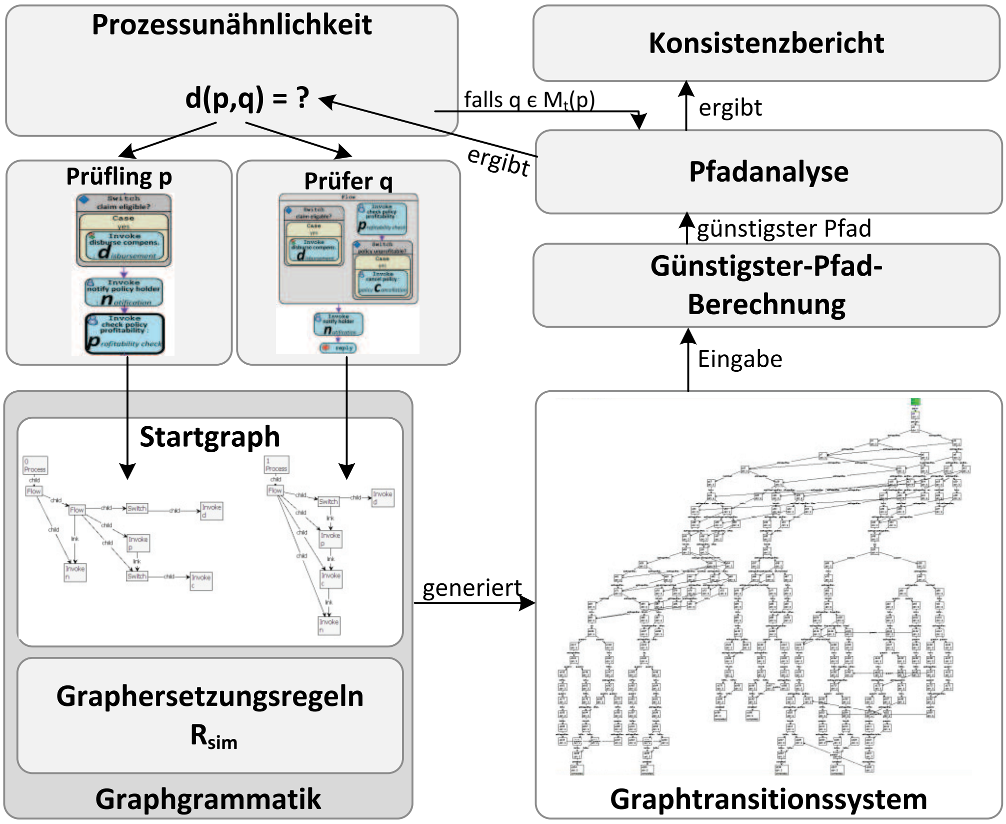


Abbildung 5.1: Ansatz für die Realisierung von Konsistenzchecks

fungen genannt, da hierbei die Aufdeckung von Inkonsistenzen zwischen Prozessmodellen gleicher Autorität im Mittelpunkt steht.

So wie Komplianzprüfungen fokussieren auch Konsistenzprüfungen Kontrollflussaspekte, d.h., es werden Unterschiede zwischen Prüfling und Prüfer bzgl. Aktivitätshäufigkeit (vergleichbar mit include-Bedingungsbeziehungen) und gegenseitiger Reihenfolge (vergleichbar mit preceded_by- und succeeded_by-Bedingungsbeziehungen) untersucht.

In diesem Kapitel wird ein Ansatz zur Realisierung von Konsistenzprüfungen erläutert. In Abschnitt 5.1 werden zunächst unterschiedliche Prozessinstanzmodelle erläutert, die später in Teilen für erläuternde Beispiele verwendet werden. Abschnitt 5.2 führt unabhängig von einer Implementierung zentrale Begriffe im Bereich von Prozessmodellähnlichkeit in Form formaler Definitionen ein. Darauf aufbauend wird in den Abschnitten 5.3ff eine Graphgrammatik-basierte Realisierung zur Ähnlichkeitsberechnung und zum detaillierten Vergleich von Prozessinstanzmodellen erläutert. Diese ist in Abbildung 5.1 zusammengefasst. Es werden zwei Prozessinstanzmodelle p (Prüfling) und q (Prüfer) in einen gemeinsamen GROOVE-Startgraphen (dualer Prozessgraph) umgewandelt. Eine Graphersetzungsregelmenge $\mathcal{R}_{sim}$ bewirkt die pseudo-simultane Ausführung der Prozessinstanzmodelle, d.h. es wird versucht, gleichgetypte Aktivitäten gleichzeitig in den Prozessinstanzmodellen auszuführen. Hieraus resultiert ein Graphtransitionssystem. Den Transitionen des Graphtransitionssystems werden unterschiedliche Kosten zugeordnet. Kosten entstehen dann, wenn simultane Ausführungen nicht möglich sind. Hieraus ist eine quantitative Berechnung der Ähnlichkeit zwischen p und q möglich, die “bilaterale Prozessunähnlichkeit” genannt wird. Falls p und q hinreichend ähnlich sind, werden Unterschiede zwischen p und q nach weiteren Analyseschritten dem Benutzer in einem Konsistenzbericht präsentiert (vgl. Abschnitt 5.6).

Ein Hauptmerkmal des Ansatzes ist, dass er auf der Ausführungssemantik (Zukunftssemantik) der Prozessinstanzmodelle aufsetzt, wie sie in Unterabschnitt 2.5.3 beschrieben wurde. Ähnlichkeiten zwischen Prozessinstanz- und -definitionsmodellen werden also nicht anhand syntaktischer Unterschiede zwischen Prozessmodellen ermittelt. Stattdessen wird das Verhalten zweier Prozessmodelle, also mögliche Ausführungspfade, miteinander verglichen. Dieses Vorgehen berücksichtigt, dass syntaktisch unähnliche Modelle durchaus ähnliches Verhalten aufweisen können [AMW06].

5.1 Struktur impliziten Prozesswissens

Die Wissensbasis, also die Menge von Prüfern für Komplianzprüfungen, besteht aus einem ggf. auch mehreren Prozesswissensmodellen in BPCL. Die Wissensbasis M_{kb} für Konsistenzprüfungen ist verglichen damit inhomogen. Sie kann in drei Partitionen unterteilt werden, die im Folgenden mit *technischen Prozessinstanzmodellklassen* bezeichnet werden. In Abbildung 5.2 sind diese technischen Prozessinstanzmodellklassen exemplarisch dargestellt. Der Lesbarkeit halber sind in den Modellen die Aktivitäten nur mit abgekürzten Aktivitätstypnamen beschriftet.

Prozessdefinitionsmodelle sind, wie in Abschnitt 2.3 erläutert wurde, syntaktische Spezialfälle von Prozessinstanzmodellen, in denen keine Elemente vorkommen, die Ausführungszustände darstellen. Sie abstrahieren also von einer konkreten Ausführung, auch von einer teilweisen. Die Prozesshistorie eines solchen Prozessinstanzmodells p ist also das leere Wort: $h(p) = \epsilon$. Sie beinhalten Informationen darüber, welche Aktivitäten in der Zukunft – mandatorisch, optional oder repetitiv – in welcher Reihenfolge ausgeführt werden müssen. Im Normalfall ist das Zukunftssemantik-definierende Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel}, \mathrm{pg}(p))}$ (vgl. Unterabschnitt 2.5.3) weitverzweigt, also mehr als eine Ausführung durch ein Prozessdefinitionsmodell abgedeckt. Das Prozessdefinitionsmodell p_3 aus Abbildung 5.2 lässt beispielsweise offen, ob die query_holder-Aktivität durchgeführt wird oder nicht.

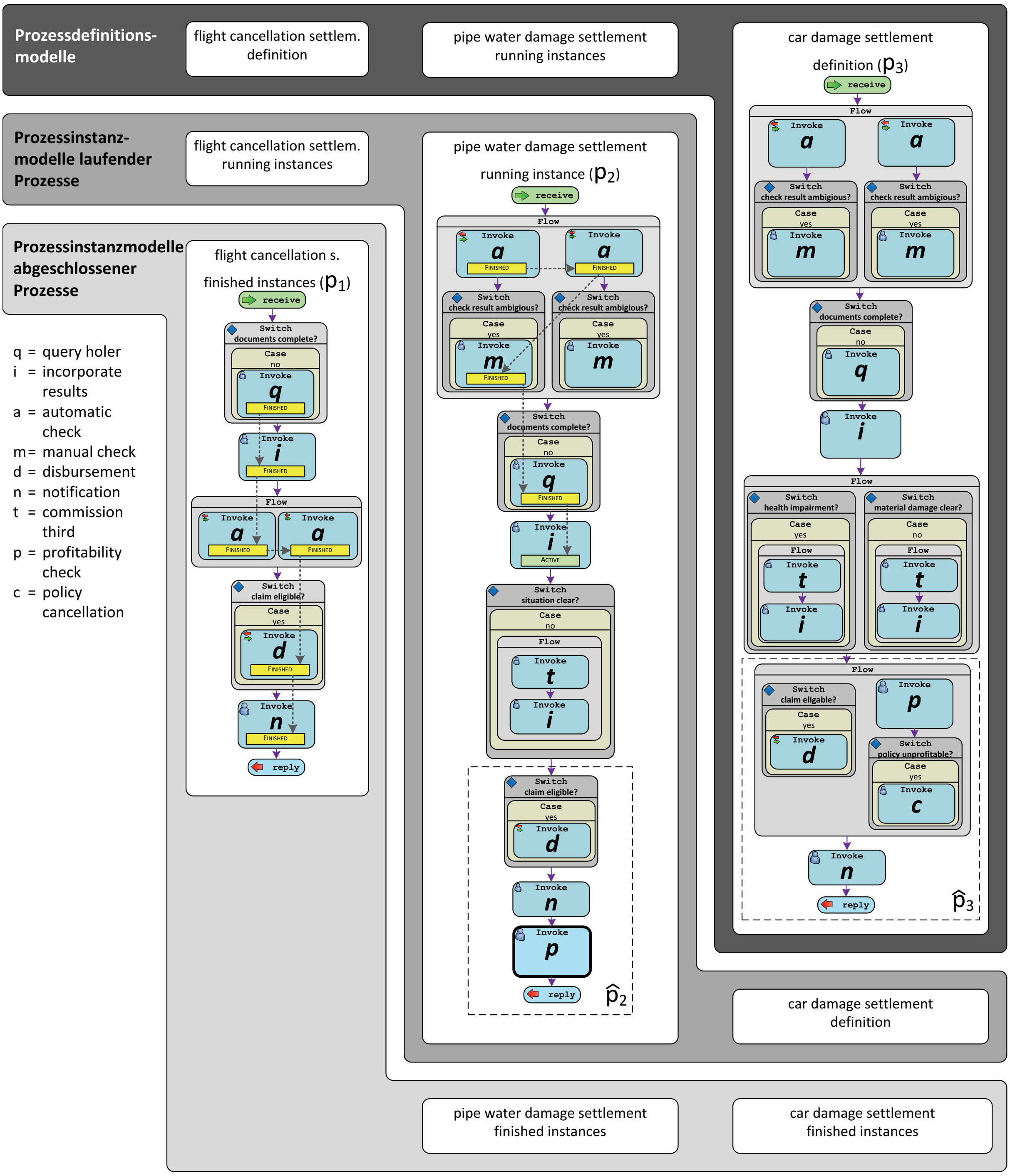


Abbildung 5.2: Technische Prozessinstanzmodellklassen

Modelle abgeschlossener Prozessinstanzen modellieren abgeschlossene Prozesse. Im Gegensatz zu Prozessdefinitionsmodellen besteht bei diesen Prozessmodellen somit keine Nichtdeterminiertheit bzgl. der Prozesszukunft mehr; alle möglichen Entscheidungen bzgl. der Prozessdurchführung wurden bereits getroffen. Das Prozessinstanzmodell p_1 aus Abbildung 5.2 ist ein Beispiel eines abgeschlossenen Prozessinstanzmodells: Alle Aktivitäten sind im Aktivitätszustand Finished; dementsprechend lautet die Prozesshistorie $h(p_1) = \mathsf{qiaadn}$. Gemäß Zukunftssemantik für Prozessinstanzmodelle gibt es keinen weiteren Zustand, den der modellierte Prozess noch annehmen kann. Das Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel}, \mathrm{pg}(p))}$ besitzt also nur noch einen Zustand.

Modelle laufender Prozessinstanzen stellen Mischformen der Modelle der zuvor genannten Klassen dar. Einerseits besitzen sie eine determinierte Vergangenheit wie Prozessinstanzmodelle abgeschlossener Prozessinstanzen, andererseits eine nicht-determinierte Zukunft wie reine Prozessdefinitionsmodelle. Beispielsweise ist der Prozess laut Prozessinstanzmodell p_2 bereits bis zur ersten *i*-Aktivität fortgeschritten. Die davor liegenden Aktivitäten wurden in einer determinierten Reihenfolge ausgeführt, nämlich $h(p_2) = \mathsf{aamqi}$. Bei den dahinter liegenden Aktivitäten ist nun aber nicht determiniert, ob diese überhaupt ausgeführt werden – abgesehen von den letzten beiden.

Alle technischen Prozessmodellklassen sind gleichsam wichtig. Dazu gehören auch laufenden Prozessinstanzen, da Prozesse bestimmter Typen langlebig sind, beispielsweise Lebensversicherungsverträge.

Da Abbildung 5.2 Gegenteiliges suggeriert, sei an dieser Stelle nochmal betont, dass Prozessinstanzmodelle dynamischen Änderungen unterliegen können. Insofern kann es sein, dass zu einem Prozessinstanzmodell kein Prozessdefinitionsmodell mit gleicher Aktivitätsstruktur existiert.

5.2 Prozesslaufbasierte Ähnlichkeit

Der paarweise Vergleich von Prozessinstanzmodellen ist wesentlich für die Nutzbarmachung impliziten Prozesswissens. Gerade bei komplexen Objekten wie Prozessinstanzmodellen ist jedoch nicht offensichtlich, welche Maßstäbe beim Vergleich angelegt werden sollen und wie das Ergebnis eines Vergleichs geartet sein soll. Dieser Abschnitt führt daher auf formalem Wege einen Ähnlichkeitsbegriff für Prozessinstanzmodelle ein; die Implementierung der Ähnlichkeitsberechnung zwischen Prozessinstanzmodellen ist Gegenstand der darauf folgenden Abschnitte.

5.2.1 Prozessläufe

Für die Semantikdefinitionen in Abschnitt 2.6 war es zweckmäßig, Prozesshistorie und -zukunft eines Prozessinstanzmodells p getrennt zu betrachten. Für die Definition von Ähnlichkeitsmaßen hingegen ist es günstiger, beides vereint zu betrachten. Hier bieten sich AP'-Sprachen an, deren Wörter ein gemeinsames Präfix haben, das die Prozesshistorie ist und deren unterschiedliche Suffixe möglichen Pfade im Zukunftssemantik-definierenden Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{bpel},\mathrm{pg}(p))}$ entsprechen. Solche Wörter seien als Prozessläufe eines Prozessinstanzmodells p bezeichnet.

Definition 5.1 (Menge der Prozessläufe) Sei M die Menge möglicher Prozessinstanzmodelle und $\mathfrak{T}_{(\mathcal{R}_{bpel},\mathrm{pg}(p))}$ das Transitionssystem, das durch die Graphersetzungsregelmenge $\mathcal{R}_{bpel}$ und den Startprozessgraphen $\mathrm{pg}(p)$ erzeugt wird (vgl. Unterabschnitt 2.5.3). Weiterhin sei $h(p)$ die Prozesshistorie von p (vgl. Unterabschnitt 2.5.2). Dann ist

$$c : M \to 2^{AP'^*};\ p \mapsto h(p) \cdot \mathcal{L}_{AP'}(\mathfrak{T}_{(\mathcal{R}_{bpel},\mathrm{pg}(p))})$$

die Abbildung, die einem Prozessinstanzmodell p eine Menge möglicher Prozessläufe zuordnet.

Beispiel 5.1 (Prozessläufe eines Prozessinstanzmodells)
Zur Verdeutlichung diene das Prozessinstanzmodell p_i rechts unten in Abbildung 5.3. Offensichtlich werden in diesem die drei Aktivitätstypennamen a, b und c verwendet, somit ist also $AP' = \{\mathsf{a,b,c}\}$. Das zu p gehörige Prozessdefinitionsmodell p_d ist über p_i abgebildet.

Die Prozessläufe des Prozessdefinitionsmodells p_d sind

$$c(p_d) = \{\mathsf{abc,ac,acb}\};$$

es sind also drei Prozessläufe möglich. Demgegenüber gibt es für p_i nur einen möglichen Prozesslauf, da hier mit $h(p_i) = \mathsf{ab}$ schon eine determinierte Historie vorliegt und c mandatorisch ausgeführt wird:

$$c(p_i) = \{\mathsf{abc}\}.$$

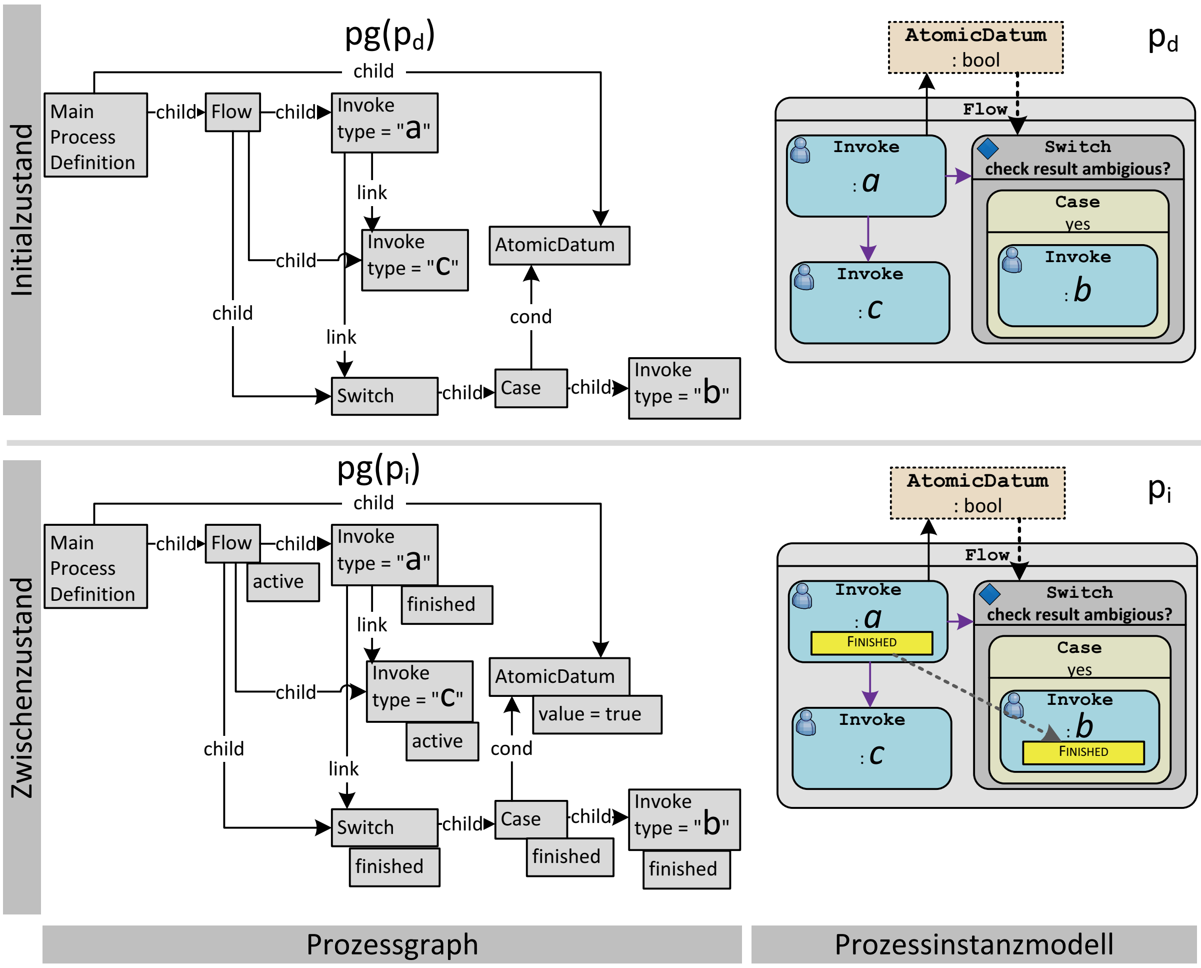


Abbildung 5.3: Prozessdefinitions- und -instanzmodell als Prozessgraph

5.2.2 Prozessunähnlichkeit

Aufbauend auf der Menge der Prozessläufe kann ein Maß für die Ähnlichkeit zwischen zwei Prozessinstanzmodellen p und q entwickelt werden. Für ein solches Maß sind zwei grundlegende Festlegungen zu treffen:

Ergebnisart Das Ergebnis eines Vergleichs kann zweiwertig sein, d.h. Ähnlichkeit wurde festgestellt oder nicht. Solche Vergleiche werden auch *qualitative* Vergleiche genannt. Demgegenüber stehen *quantitative* Vergleiche, deren Ergebnisse skalar sind und Aussagen über das Ausmaß der Ähnlichkeit zweier Prozessinstanzmodelle treffen.

Ähnlichkeitsbegriff Die Ähnlichkeit zweier Prozessinstanzmodelle kann mit unterschiedlicher Strenge beurteilt werden. Bei unterschiedlichen Begriffen kann für zwei Prozessinstanzmodelle p und q qualitativ Ähnlichkeit gegeben sein bzw. quantitativ maximale Ähnlichkeit vorliegen, bei Verwendung eines strengeren Begriffs für die gleichen Modelle jedoch nicht. Ein strenger Begriff von Ähnlichkeit liegt beispielsweise vor, wenn $c(p) = c(q)$ für zwei Prozessinstanzmodelle p und q gefordert wird, also die gleichen Prozessläufe erzeugt werden müssen. Eine abschwächte Form liegt vor, wenn nur $c(p) \cap c(q) \neq \emptyset$ gefordert wird, also nur mindestens ein gemeinsamer Prozesslauf gefordert wird. Es existieren noch weitere Ähnlichkeitsbegriffe, die auf anderen Formalismen beruhen und daher nicht an dieser Stelle sondern erst in Abschnitt 5.8 diskutiert werden.

Zum Zweck der Nutzung impliziten Prozesswissens werden die genannten Festlegungen folgendermaßen getroffen: Das Ergebnis ist quantitativ, da ein rein qualitatives Ergebnis für den genannten Zweck zu grob ist. Ferner wird der oben beschriebene abgeschwächte Ähnlichkeitsbegriff verwendet, da bereits ein gemeinsamer Prozesslauf zwischen zwei Prozessinstanzmodellen in der betrachteten Menge M_{kb} eher die Ausnahme als die Regel darstellt.

Das Ergebnis eines Prozessinstanzmodellvergleichs ist in dieser Arbeit eine natürliche Zahl. Ein Ergebnis von Null bedeutet dabei eine maximale Ähnlichkeit bezogen auf den angelegten Ähnlichkeitsbegriff. Jedes Ergebnis größer als Null bedeutet, dass die verglichenen Prozessinstanzmodelle keine gemeinsamen Prozessläufe besitzen. Da ein großes Ergebnis eine geringere Ähnlichkeit bedeutet, ist es intuitiver, von Prozess*un*ähnlichkeit zu sprechen. Außerdem ist zu betonen, dass jeweils immer genau zwei Prozessinstanzmodelle miteinander verglichen werden, der Vergleich also bilateral ist. Das im Folgenden verwendete Ähnlichskeitsmaß wird daher *bilaterale Prozessunähnlichkeit* genannt.

Die bilaterale Prozessunähnlichkeit basiert auf einem Maß, dass die Ähnlichkeit von Prozessläufen repräsentiert. In dieser Arbeit wird dafür die Levenshtein-Distanz [Lev66] verwendet. Die Levenshtein-Distanz ist auf Zeichenketten definiert und kann daher auch für Prozessläufe verwendet werden. Sie misst den Editierabstand zweier Wörter (Zeichenketten) über die Anzahl der Einfüge-, Lösch- und Ersetzungsoperationen, die mindestens notwendig sind, um die Identität eines Wortes zu einem anderen herzustellen, ist also eine Funktion

$$l : \mathcal{L} \times \mathcal{L} \rightarrow \mathbb{N}_0,$$

wobei $\mathcal{L}$ eine Sprache ist, deren Elemente Zeichenketten sind.

Definition 5.2 (Bilaterale Prozessunähnlichkeit) Sei M die Menge aller möglichen Prozessinstanzmodelle und $c(p)$ sowie $c(q)$ die Menge aller Prozessläufe der Prozessinstanzmodelle p bzw. q. Sei ferner l die Levenshtein-Distanz zwischen zwei Prozessläufen. Die bilaterale Prozessunähnlichkeit ist definiert als Funktion

$$d : M \times M \to \mathbb{N}_0;\; (p,q) \mapsto \min_{\substack{t_p \in c(p),\\ t_q \in c(q)}} \{l(t_p,t_q)\}.$$

Insbesondere gilt, $d(p,q) = 0$ genau dann, wenn $c(p) \cap c(q) \neq \emptyset$ ist, d.h. die minimale Prozessunähnlichkeit (die maximale Prozessähnlichkeit) ist bereits erreicht, wenn ein gemeinsamer Prozesslauf existiert.

Die reine quantitative Angabe der Unähnlichkeit zweier Prozessinstanzmodelle ist für Prozessbeteiligte von begrenztem Nutzen. Sie kann jedoch verwendet werden, um eine Teilmenge in M_{kb} zu identifizieren, die nur Prozessinstanzmodelle enthält, die hinreichend ähnlich zu einem Prüfling p sind. Diese Teilmenge ist dann eine Kandidatenmenge von Prüfern, die sinnvollerweise einem detaillierten Vergleich mit q unterzogen werden können. Alle übrigen Prozessinstanzmodelle gelten dann als so unähnlich zu p, dass eine detaillierte Analyse keine nützlichen Ergebnisse liefern kann.

Definition 5.3 (t-ähnliche Prozessinstanzmodelle) Für ein Prozessinstanzmodell p sei die Menge aller t-ähnlichen Prozessinstanzmodelle aus der Grundmenge M_{kb} definiert als

$$M_t(p) := \{q \in M_{kb} \mid d(p,q) \leq t\}.$$

5.3 Pseudo-simultane Prozessdurchführung

Die Definitionen des letzten Abschnitts geben keinerlei Aufschluss darüber, wie einerseits die bilaterale Prozessunähnlichkeit d und die darauf aufbauende Menge $M_t(p)$ prinzipiell berechnet werden können. Diese Fragestellung ist Gegenstand dieses Abschnitts.

Ein naiver Algorithmus zur Berechnung der bilateralen Prozessunähnlichkeit $d(p,q)$ zweier Prozessinstanzmodelle p und q könnte folgendermaßen lauten:

1. Erzeuge die Mengen $c(p)$ und $c(q)$.

2. Finde ein Paar $(t_p,t_q) \in c(p) \times c(q)$ mit minimalem $l(t_p,t_q)$.

Natürlich ist diese Vorgehensweise aus mehreren Gründen nicht praktikabel: Die Erzeugung von $c(p)$ und $c(q)$ ist hinsichtlich Zeit- und Platzkomplexität gerade bei Prozessinstanzmodellen mit komplizierten Link-Strukturen schwierig. Bei Prozessinstanzmodellen mit Schleifen (`While`-Aktivitäten) ist die Erzeugung sogar unmöglich, da Schleifen beliebig häufig durchlaufen werden können, somit also die betreffende Menge von Prozessläufen unendlich groß werden kann. Des Weiteren muss zur Minimumsberechnung jeder Prozesslauf aus $c(p)$ mit jedem aus $c(q)$ verglichen werden, d.h. es müssen $|c(p)| \cdot |c(q)|$ Levenshtein-Distanzen verglichen werden.

Statt des naiven Vorgehens wird im Weiteren ein Ansatz verfolgt, der auf den aus Abschnitt 2.5 bekannten GROOVE-Graphersetzungsregeln und -Graphtransitionssystemen beruht. Ausgangspunkt ist die Graphersetzungsregelmenge $\mathcal{R}_{bpel}$, über die die Zukunftssemantik von Prozessinstanzmodellen definiert wurde. Aus später erörterten Gründen wird diese Regelmenge zu einer Regelmenge $\mathcal{R}_{sim}$ abgewandelt.

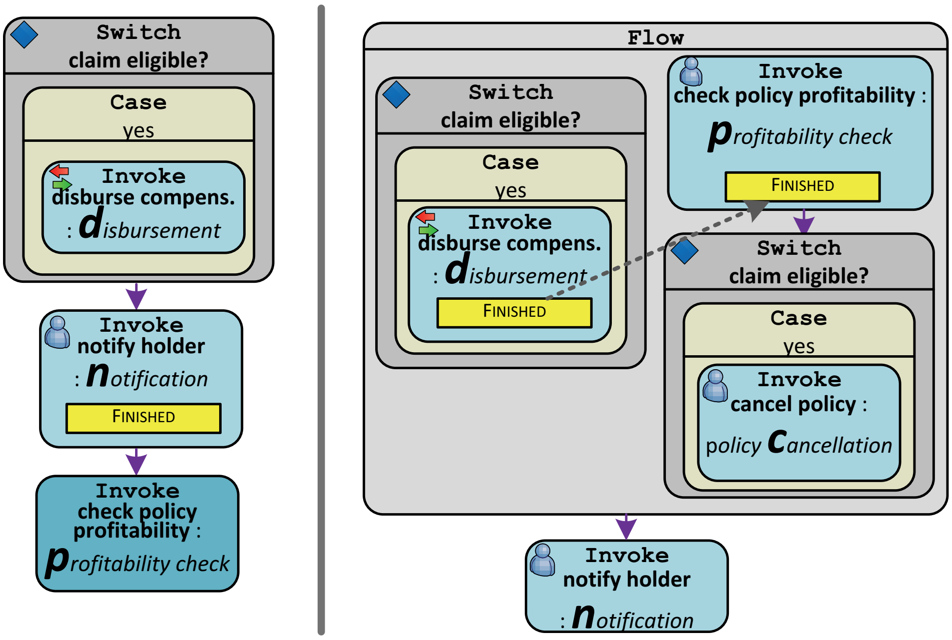
5.3.1 Dualer Prozessgraph

Die Veränderung der Regelmenge $\mathcal{R}_{sim}$ gegenüber $\mathcal{R}_{bpel}$ betrifft zunächst die Form der Graphen, die $\mathcal{R}_{sim}$ transformiert und die die Zustände des erzeugten Transitionssystems bilden. Die Form sei an einem Beispiel erläutert:

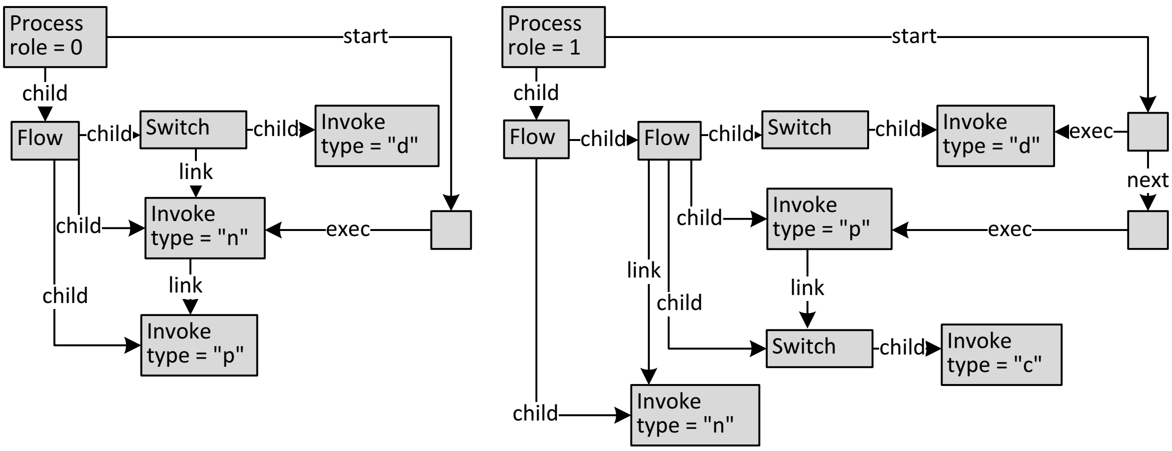
Beispiel 5.2 (Dualer Prozessgraph) In Abbildung 5.4(a) sind zwei Prozessinstanzmodelle $\hat{pi}_2$ und $\hat{pi}_3$ abgebildet. Beide sind Fragmente der Prozessinstanzmodelle p_2 und p_3 aus Abbildung 5.2, allerdings mit abweichenden Ausführungszuständen: In $\hat{pi}_2$ wurde die *d*-Aktivität nicht ausgeführt. Die Ausführung ist bis zur *n*-Aktivität fortgeschritten. In $\hat{pi}_3$ wurde die *d*-Aktivität ausgeführt und zwar vor der *p*-Aktivität. In Abbildung 5.4(b) ist der duale Prozessgraph $\text{pg}'(\hat{pi}_2, \hat{pi}_3)$ dargestellt.

Die Unterschiede zu den Prozessgraphen, die sich durch die Abbildung pg ergeben (vgl. Unterabschnitt 2.5.1), sind wie folgt:

- Offensichtlich werden immer genau zwei Prozessinstanzmodelle durch einen dualen Prozessgraphen widergespiegelt, die jedoch nicht in Aufrufbeziehung zueinander stehen (vgl. Beispiel 2.10).
- Aktivitätszustände sind zunächst nicht enthalten. Sie werden später, d.h. bei der Anwendung entsprechender Regeln in $\mathcal{R}_{sim}$, nicht in separaten Knoten gespeichert, sondern direkt als Knotenbeschriftung der betreffenden Aktivität.



(a) Prozessinstanzmodelle $\hat{pi}_2$ und $\hat{pi}_3$



(b) Dualer Prozessgraph $\mathrm{pg}'(\hat{pi}_2, \hat{pi}_3)$

Abbildung 5.4: Prozessinstanzmodelle und entsprechender dualer Prozessgraph

- Neben der Kontrollflussdefinition ist in dualen Prozessgraphen auch die Prozesshistorie enthalten. Die Prozesshistorie wird gebildet aus einer über next-Kanten linear verketteten Liste, deren Knoten exec-Kanten zur jeweils ausgeführten Aktivität besitzen. Die start-Kante markiert den Anfang der Historie. Die zusätzliche Struktur wird in $\mathcal{R}_{sim}$ berücksichtigt. Sie erzwingt die Ausführung der durch exec-Kanten referenzierten Aktivitäten in der durch next-Kanten gegebenen Reihenfolge. Im zu $\hat{pi}_3$ gehörenden Teil des dualen Prozessgraphen in Abbildung 5.4(b) wird beispielsweise die Ausführung von zunächst der *d*- und dann der *p*-Aktivität erzwungen. Im zu $\hat{pi}_2$ gehörenden Teil wird durch die Prozesshistorie indirekt erzwungen, dass die *d*-Aktivität nicht ausgeführt wird, jedoch die *n*-Aktivität.
- Prozessvariablen sind in dualen Prozessgraphen nicht enthalten.
- Dementsprechend wird bei `Switch`- und `While`- Aktivitäten unabhängig von einer aktuellen Belegung einer Prozessvariablen entschieden, ob ein und ggf. welcher Zweig aktiviert wird bzw. ob eine (weitere) Iteration stattfindet.
- Somit ist die explizite Modellierung von `Case`-Aktivitäten in dualen Prozessgraphen nicht notwendig.
- Das Knotenattribut role identifiziert die einzelnen Prozessgraphen im dualen Prozessgraphen. Per Konvention ist der Wert des Attributs 0 bzw. 1, wenn der Prozessgraph zum Prüfling bzw. Prüfer gehört.

Die Vereinfachungen bezwecken eine Speicherplatz- und Laufzeitoptimierung bei der Erzeugung des Graphtransitionssystems durch die Regelmenge $\mathcal{R}_{sim}$. Die Vereinfachungen im dualen Prozessgraphen bewirken direkt eine Verkleinerung der Größe eines einzelnen Zustands im Transitionssystem, indirekt und im Zusammenhang mit der Regelmenge $\mathcal{R}_{sim}$ eine Reduktion der Zustandsanzahl, da beispielsweise Zustände wegfallen, die sich nur durch unterschiedliche Ausführungszustände einer `Case`-Aktivität unterscheiden.

5.3.2 Die Graphersetzungsregelmenge $\mathcal{R}_{sim}$

Die Graphersetzungsregelmenge $\mathcal{R}_{sim}$ wird im Folgenden erläutert, indem die wichtigsten Regeln anhand ihrer Wirkung in einem Transitionssystemausschnitt demonstriert werden. Der Ausschnitt stammt aus dem Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(\hat{p}_2,\hat{p}_3))}$, das in Abbildung 5.5 gezeigt ist. Die Modelle $\hat{p}_2$ und $\hat{p}_3$ sind wieder die in Abbildung 5.2 markierten Fragmente der Prozessinstanzmodelle p_2 bzw. p_3 allerdings ohne abweichende Ausführungszustände.

Die Notation der Graphzustände ist aus Platzgründen und der Lesbarkeit halber an die konkrete Syntax der Prozessmodelle $\hat{p}_2$ und $\hat{p}_3$ angelehnt. Es

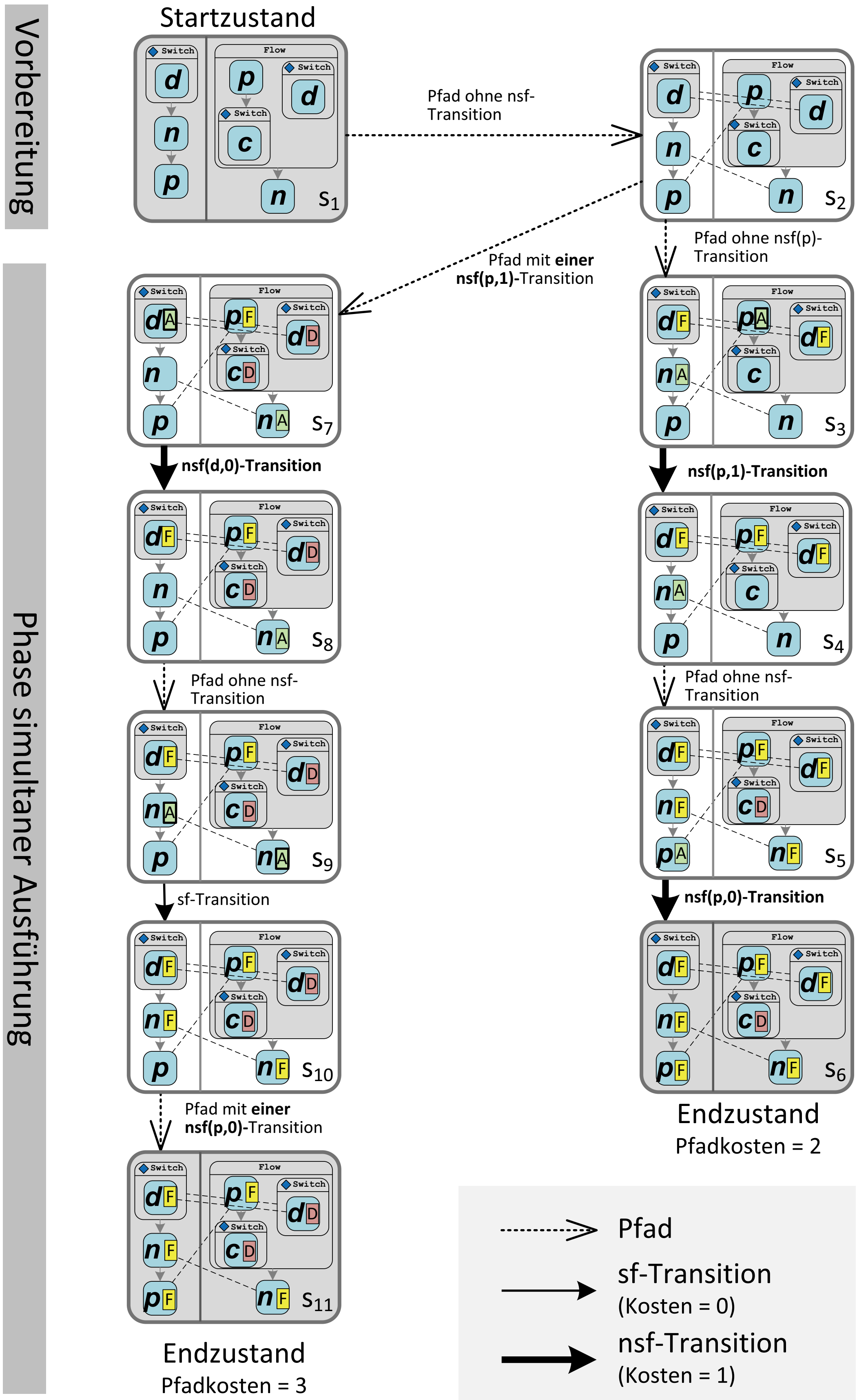


Abbildung 5.5: Ausschnitt aus Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(\hat{p}_2,\hat{p}_3))}$

ist zu beachten, dass die Invoke-Aktivitäten nur mit dem Anfangsbuchstaben ihres Typs beschriftet sind. Die Beschriftungen entsprechen damit den in Abbildung 5.2 hervorgehobenen Buchstaben.

Neben der Notation ist das Transitionssystem insoweit vereinfacht und ausschnitthaft, als dass bei Weitem nicht alle Pfade abgebildet sind, sondern nur zwei ausgewählte. Des Weiteren sind in den Pfaden einige Zwischenzustände ausgelassen worden.

Ein Transitionssystem, das durch $\mathcal{R}_{sim}$ erzeugt wird, zerfällt in zwei Phasen: die Vorbereitungsphase und die Ausführungsphase. Die Trennung der beiden Phasen ist eindeutig, d.h. das erzeugte Transitionssystem kann so in zwei Teiltransitionssysteme aufgeteilt werden, dass beide nur einen gemeinsamen Zustand haben, der im Transitionssystem der Vorbereitungsphase der alleinige Endzustand ist und im Transitionssystem der simultanen Ausführung der Startzustand. Im Beispiel von Abbildung 5.5 ist das der Zustand s_2.

Vorbereitungsphase In der Vorbereitungsphase werden Vorbereitungen getroffen, die die Phase der simultanen Ausführung insoweit optimieren, als dass das betreffende Teiltransitionssystem stark verkleinert wird. Es ist für die Graphersetzungsregeln der Ausführungsphase günstig, typgleiche Aktivitäten bereits im Vorfeld explizit zu Paaren zusammenzufassen. In Abbildung 5.5 ist dies auf dem Pfad von s_1 zu s_2 der Fall. Die Paarbildung geschieht mittels expliziter, neuer pair-Kanten, die in der Abbildung durch strichlierte Linien dargestellt sind. Die Paarbildung von Invoke-Aktivitäten – also von Knoten, die mit Invoke beschriftet sind – wird durch eine mit pairInv benannte Graphersetzungsregel durchgeführt, die von komplexen Aktivitäten durch eine mit pairCplx benannte. Die Paarbildung in Abbildung 5.5 ist insoweit ein Spezialfall, als dass jeder Typ auf jeder Seite, d.h. jedem der beiden Prozessgraphen im dualen Prozessgraphen, maximal einmal auftaucht. Im allgemeinen Fall, in dem ein Typ *x* in einem Prozessgraphen n-mal und im anderen m-mal vorkommt, werden $n \cdot m$ pair-Kanten eingefügt.

Phase pseudo-simultaner Ausführung Während der pseudo-simultanen Ausführung werden die beiden, durch den dualen Prozessgraphen dargestellten Prozessinstanzmodelle ausgeführt. Es werden dazu Regeln $\mathcal{R}_{sim}$ vergleichbar mit $\mathcal{R}_{bpel}$ aus Unterabschnitt 2.5.3 angewendet. Regeln in $\mathcal{R}_{sim}$ unterscheiden sich dabei insofern von denen in $\mathcal{R}_{bpel}$, als dass soweit möglich, aktive, typgleiche und somit gepaarte Aktivitäten aus beiden Prozessmodellen in einem Schritt, also durch eine Regelanwendung, finalisiert werden.

Simultane Finalisierung (sf) Die Regel sf ("simultane Finalisierung") ändert den Zustand zweier Aktivitäten gleichzeitig auf Finished. Im Beispiel von

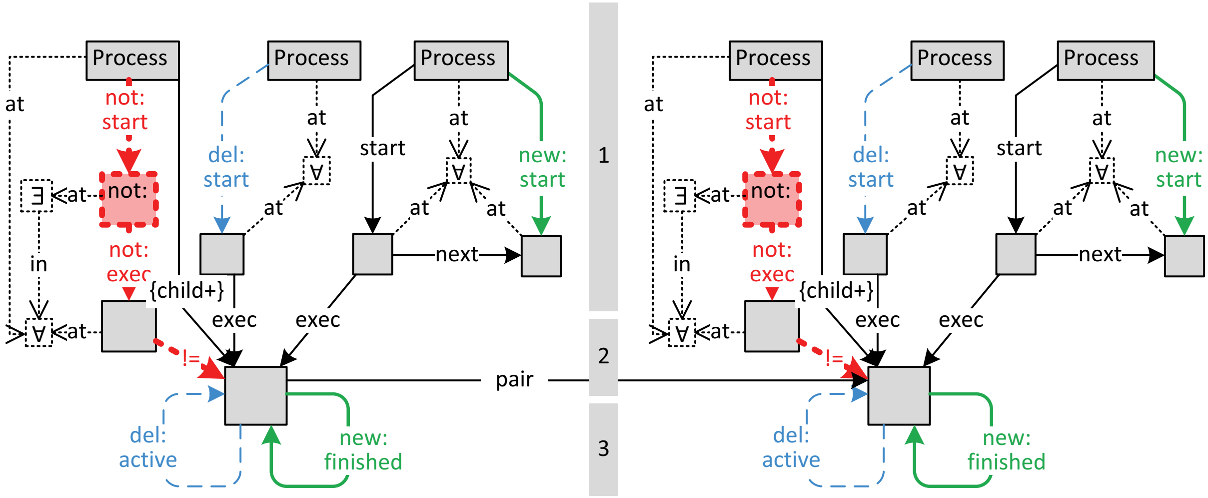


Abbildung 5.6: Graphersetzungsregel sf $\in \mathcal{R}_{sim}$

Abbildung 5.5 wird die Regel beispielsweise beim Übergang von s_9 zu s_{10} angewendet. Bei diesem Übergang werden die *n*-Aktivitäten simultan von Active (A) auf Finished (F) gesetzt. Die Abbildung 5.6 zeigt die Regel sf. Die darin ausgedrückten Anwendbarkeitsbedingungen sind die Folgenden:

1. Die Prozesshistorie wird berücksichtigt. Das bedeutet, dass eine Aktivität in einem Prozess nicht finalisiert werden kann, wenn laut Prozesshistorie erst eine andere, frühere Aktivität finalisiert werden muss. Die Berücksichtigung der ggf. vorhandenen Prozesshistorien beim Aufbau des Transitionssystems ist wichtig. Schließlich ist bei Betrachtung eines Prozessinstanzmodells p irrelevant, wie bereits abgelaufene Prozessteile laut Kontrollflussdefinition auch anders hätten ausgeführt werden können, sondern nur, wie sie tatsächlich ausgeführt wurden. Dieser Gedanke folgt der einfachen Erkenntnis, dass die Prozesshistorie unveränderlich ist.
2. Die Aktivitäten müssen in der Vorbereitungsphase gepaart worden sein, was durch die pair-Kante gefordert wird.
3. Die Aktivitäten müssen im Zustand Active sein.

Nicht-simultane Finalisierung (nsf) Die Regel sf ist nur bedingt für aktive Aktivitäten anwendbar. Insbesondere das Muster ② in Abbildung 5.6 passt häufig nicht im Kontext aktiver Aktivitäten, d.h. es gibt häufig kein Paar aktiver und gepaarter und somit gleichgetypter Aktivitäten. Die Regel nsf (Abk. für "nicht-simultane Finalisierung") gleicht der Regel sf bis auf den Teil ②, d.h. sie gleicht dem linken (oder dem identischen rechten) Teil der Regel sf und fordert somit nicht das Vorhandensein eines Gegenstücks für eine zu finalisierende aktive Aktivität. Im Transitionssystem aus Abbildung 5.5 wird die Regel nsf beispielsweise beim Übergang von s_7 zu s_8 angewendet. Die bewirkte Veränderung ist die nicht-simultane Finalisierung der *d*-Aktivität (links oben in den Zuständen).

Aus Gründen der Einfachheit wurde das Beispieltransitionssystem aus Abbildung 5.5 auf Basis von Prozessinstanzmodellen $\hat{p}_2$ und $\hat{p}_3$ erzeugt, die keine Ausführungszustände besitzen und somit den Spezialfall von Prozessdefinitionsmodellen darstellen. Die Berücksichtigung der Prozesshistorie durch die Regeln sf und nsf kann hierdurch nicht demonstriert werden. Dies wird anhand eines – wiederum ausschnitthaften – Transitionssystems $\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(\hat{pi}_2, \hat{pi}_3))}$ nachgeholt, also auf Basis der Prozessinstanzmodelle aus Abbildung 5.4.

Beispiel 5.3 (Transitionssystem mit Prozesshistorie) Der Ausschnitt aus dem Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(\hat{pi}_2, \hat{pi}_3))}$ ist in Abbildung 5.7 dargestellt. Der Zustand s_2 ist gleich dem gleichnamigen Zustand im Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(\hat{p}_2, \hat{p}_3))}$ aus Abbildung 5.5 und die Zustände s_i' sind gleich den Zuständen s_i bis auf die Tatsache, dass die rechte *d*-Aktivität deaktiviert statt finalisiert ist. Damit ist im Transitionssystem die Historie in den Prozessinstanzmodellen $\hat{pi}_2$ und $\hat{pi}_3$ berücksichtigt (vgl. Abbildung 5.4). Diese erzwingt insbesondere Folgendes:

1. Rechts in $\hat{pi}_3$ muss die *d*-Aktivität vor der *p*-Aktivität ausgeführt werden. Der Zustand s_x kann also gar nicht in der Zustandsmenge des Transitionssystems sein.

2. Die *d*-Aktivität rechts in $\hat{pi}_3$ muss ausgeführt werden, die *d*-Aktivität links in $\hat{pi}_2$ darf nicht ausgeführt werden, was indirekt aus der Historie von $\hat{pi}_2$ hervorgeht. Demzufolge kann s_y ebenfalls nicht in der Zustandsmenge enthalten sein.

Letzteres hat zur Folge, dass die *d*-Aktivitäten nicht simultan, also durch Anwendung von sf finalisiert werden können. Stattdessen muss die rechte *d*-Aktivität in jedem Fall, d.h. auf jedem Pfad im Transitionssystem nicht-simultan, also mittels nsf finalisiert werden. Dementsprechend verteuert sich der günstigste Pfad in $\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(\hat{pi}_2, \hat{pi}_3))}$ gegenüber dem in $\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(\hat{p}_2, \hat{p}_3))}$ um eins auf drei, $\hat{pi}_3$ und $\hat{pi}_2$ sind also unähnlicher als $\hat{p}_3$ und $\hat{p}_2$, d.h. $d(\hat{pi}_3, \hat{pi}_2) > 2 = d(\hat{p}_3, \hat{p}_2)$.

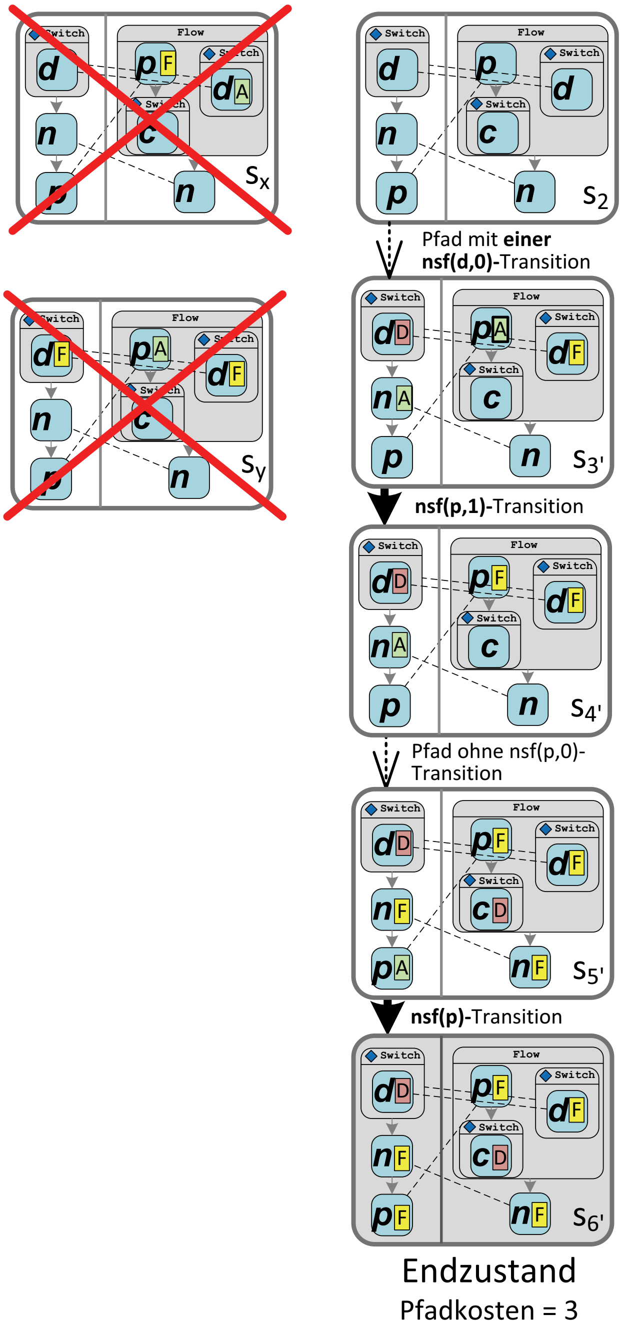


Abbildung 5.7: Pfad und ausgeschlossene Zustände

5.4 Günstigste Pfade im Graphtransitionssystem

Endzustände in durch $\mathcal{R}_{sim}$ erzeugten Transitionssystemen sind solche Zustände, in denen alle Aktivitäten entweder deaktiviert (D) oder finalisiert (F) sind. Im Transitionssystem-Ausschnitt aus Abbildung 5.5 sind das die Zustände s_{11} und s_6. Am Transitionssystem kann durch Betrachtung der Pfade ausgewertet werden, wie viele nsf-Anwendungen nötig sind, um zwei Prozessinstanzmodelle pseudo-simultan durchzuführen. Falls ein gemeinsamer Prozesslauf $a_1 \ldots a_n$ existiert, gibt es einen Pfad, auf dem es keine nsf-Transition gibt und sf-Transitionen in der Reihenfolge sf(a_1) ... sf(a_n) vorkommen.

Im Beispiel von Abbildung 5.5 sind zwei Pfade dargestellt. Auf dem nach links abzweigenden Pfad sind drei nsf-Transitionen vorhanden. Hier wird im linken Prozessteilgraphen unnötigerweise die *d*-Aktivität nicht ausgeführt, obwohl sie simultan zur *d*-Aktivität im rechten Prozessteilgraphen durchgeführt werden könnte. Auf dem nach rechts abzweigenden Pfad ist dies nicht der Fall. Dementsprechend sind hier die Kosten niedriger.

Offensichtlich gibt es Pfade im Transitionssystem, deren Kosten unnötig groß sind. Deswegen kann nicht irgendein Pfad zur Berechnung der bilateralen Prozessunähnlichkeit herangezogen werden. Vielmehr ist die bilaterale Prozessunähnlichkeit d gleich den Kosten des günstigsten Pfades im Transitionssystem:

$$d(\hat{p}_2, \hat{p}_3) = 2.$$

Tatsächlich wird die Unähnlichkeit auch zwischen den kompletten Prozessinstanzmodellen p_2 und p_3 nur durch die Unähnlichkeit zwischen $\hat{p}_2$ und $\hat{p}_3$ verursacht, d.h. es ist ebenfalls

$$d(p_2, p_3) = 2.$$

Der günstigste Pfad wird mit einer Variante des Dijkstra-Algorithmus [Dij59] berechnet, der in [Fis09] detailliert beschrieben ist.

Über die bilaterale Prozessunähnlichkeit d kann die Menge t-ähnlicher Prozessinstanzmodelle q für ein Prozessinstanzmodell p berechnet werden. Sei die Menge betrachteter Prozessinstanzmodelle $M_{kb} = \{p_1, p_2, p_3\}$ aus Abbildung 5.2. Dann ist für einen Prüfling p_2 die Menge 2-ähnlicher Prüfer $M_2(p_2) = \{p_2, p_3\}$. Das Prozessinstanzmodell p_1 ist nicht enthalten, da $d(p_2, p_1) = 3 > 2$ ist. Der Prüfling p_2 selbst ist in der Menge trivialerweise enthalten.

5.5 Detailanalyse günstigster Pfade

Mit der Menge t-ähnlicher Prozessinstanzmodelle könnten bereits qualitative Aussagen über die Ähnlichkeit von Prozessinstanzmodellen getroffen werden. Solche Aussagen sind für Prozessbeteiligte von begrenztem Nutzen. Die Menge kann jedoch auch als Basis für detaillierte Analysen der Prozessunähnlichkeit verwendet werden. Ergebnis dieser detaillierten Analysen ist ein Konsistenzbericht, der letztlich dem Prozessbeteiligten präsentiert wird.

In den durch $\mathcal{R}_{sim}$ erzeugten Transitionssystemen sind die durch die Anwendung der Regel nsf entstandenen Transitionen unter anderem mit dem Typnamen der finalisierten Aktivität beschriftet. Die Transition von s_7 zu s_8 in Abbildung 5.5 ist beispielsweise mit nsf(d,0) beschriftet. Hierbei ist d ein Transitionsparameter und beinhaltet den Typnamen der Aktivität, die durch die Regel nsf finalisiert wurde. Der zweite Parameter 0 ist der Wert des role-Attributs des Prozessgraphen, in dem die nsf-Regel angewendet wurde. Anhand der Transitionsparameter können Detailunterschiede zwischen zwei Prozessinstanzmodelle p und q mit $d(p,q) > 0$ ermittelt werden, was im Folgenden erläutert wird.

Definition 2.11 ordnet einem Transitionssystem eine Sprache von Wörtern zu, deren Symbole atomare Aussagen sind. Relevant sind für die detaillierte Pfadanalyse nur die Häufigkeiten und Reihenfolgen von nsf-Transitionen, also atomare Aussagen, die aufgrund der Anwendbarkeit der nsf-Graphersetzungsregel gelten. Es ist für die weitere Formalisierung günstig, eine Sprache zu definieren, deren Symbole nsf zzgl. eines Parameters für den Typnamen der finalisierten Aktivität und eines für die Prozessrolle (0 für Prüfling und 1 für Prüfer) sind.

Definition 5.4 (NSF-Sprache) Sei

$$NSF := \{\mathrm{nsf}(x,k) \mid x \text{ ist ein Aktivitätstypname und } k \in \{0,1\}\}$$

ein auf atomare Aussagen nsf(x,k) begrenztes Alphabet. Dann sei gemäß Definition 2.11 für zwei Prozessinstanzmodelle p und q

$$\mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))})$$

die NSF-Sprache von $\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))}$.

Beispiel 5.4 (Beispielworte in NSF-Sprache) Es gilt Folgendes für das Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(\hat{p}_2,\hat{p}_3))}$ aus Abbildung 5.5

- nsf(p,1)nsf(d,0)nsf(p,0) $\in \mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(\hat{p}_2,\hat{p}_3))})$, da dies die Reihenfolge der nsf-Transitionen im linken Pfad in Abbildung 5.5 ist.
- nsf(p,1)nsf(p,0) $\in \mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(\hat{p}_2,\hat{p}_3))})$, da dies die Reihenfolge der nsf-Transitionen im rechten Pfad in Abbildung 5.5 ist.

Wie zuvor erläutert, sind allerdings im Transitionssystem auch Pfade vorhanden, die unnötig viele nsf-Transitionen beinhalten. Relevant für die Detailanalyse sind wiederum nur die günstigsten Pfade, was in $\mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))})$ den kürzesten Wörtern entspricht.

Definition 5.5 (MinNSF-Sprache) Sei für zwei Prozessinstanzmodelle p und q die NSF-Sprache $\mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))})$ wie in Definition 5.4 definiert. Dann ist

$$\mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))})$$
$$:=$$
$$\{w \in \mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))}) \mid \neg\exists u \in \mathcal{L}_{NSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))}) \ : \ |u| < |w|\}$$

die auf kürzeste Wörter beschränkte NSF-Sprache, die das Transitionssystem $\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))}$ erzeugt.

Aus den Wörtern in $\mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))})$ lassen sich Details über die Unähnlichkeit zweier Prozessinstanzmodelle p und q ablesen.

Beispiel 5.5 (MinNSF-Sprache für $\hat{p}_2$ und $\hat{p}_3$) Die MinNSF-Sprache für die Prozessinstanzmodelle $\hat{p}_2$ und $\hat{p}_3$ ist

$$\mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(\hat{p}_2,\hat{p}_3))}) = \{\text{nsf(p,1)nsf(p,0), nsf(n,0)nsf(n,1)}\}.$$

Es gibt also zwei kürzeste Wörter; im ersten wird die *p*-Aktivität, im zweiten die *n*-Aktivität nicht-simultan finalisiert.

5.5.1 Ausführungshäufigkeiten

Eine Aktivität vom Typ x kann auf den günstigsten Pfaden unterschiedlich häufig nicht-simultan ausgeführt werden. Dies ist der Fall, wenn für ein

$$w \in \mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(p,q))})$$

das Folgende gilt:

$$|w|_{nsf(x,0)} \neq |w|_{nsf(x,1)}.$$

Insbesondere wird x im Prüfling k-mal häufiger ausgeführt, wenn

$$|w|_{nsf(x,0)} - |w|_{nsf(x,1)} = k > 0$$

gilt bzw. im Prüfer k-mal häufiger, wenn

$$|w|_{nsf(x,1)} - |w|_{nsf(x,0)} = k > 0$$

gilt. Für die Ausführungshäufigkeiten ist also die Differenz zwischen den Vorkommen von nsf(x,0) und nsf(x,1) relevant.

Beispiel 5.6 (Ausführungshäufigkeiten in $\hat{p}_2$ und $\hat{p}_3$) In der MinNSF-Sprache des Transitionssystems aus Abbildung 5.5 sind keine Häufigkeitsunterschiede vorhanden. Zwar existiert eine ungepaarte c-Aktivität, diese wird jedoch nur bedingt ausgeführt und wird daher auf dem günstigsten Pfad deaktiviert. Wäre sie erstens eine mandatorische Aktivität, also insbesondere nicht in eine `Switch`-Aktivität eingeschachtelt oder zweitens Bestandteil der Prozesshistorie, so wäre nsf(c,1) ein Symbol in jedem Wort der NSF-Sprache und $|w|_{nsf(c,1)} - |w|_{nsf(c,0)} = 1$.

Von weiterem Interesse sind die Aktivitätstypnamen, deren Aktivitäten in den Prozessinstanzmodellen unterschiedlich häufig ausgeführt werden. Es ist zweckmäßig, hierfür Mengen zu definieren:

Definition 5.6 (Ausführungshäufigkeiten) Seien p und q zwei Prozessinstanzmodelle, $k \in \mathbb{N}_0$ und $i \in \{0,1\}$. Sei ferner AP die Menge von Aktivitätstypnamen, die in p und q vorkommen. Dann ist

$$\mathcal{H}_k^i(p,q)$$
$$:=$$
$$\{x \in AP \mid \exists w \in \mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim}, \mathrm{pg}'(p,q))}) \;:\; |w|_{nsf(x,i)} - |w|_{nsf(x,1-i)} = k\}.$$

Die Menge $\mathcal{H}^i(p,q)$ abstrahiert vom Ausmaß des Unterschieds der Ausführungshäufigkeiten:

$$\mathcal{H}^i(p,q) := \bigcup_{n>0} \mathcal{H}^i_n(p,q).$$

Die Menge $\mathcal{H}^0_2(p,q)$ ist beispielsweise die Menge der Typnamen solcher Aktivitäten, die im Prüfling p ($i = 0$) zweimal ($k = 2$) häufiger (nicht-simultan) als im Prüfer q finalisiert werden. Die Menge $\mathcal{H}^1_1$ ist die Menge der Typnamen solcher Aktivitäten, die im Prüfer q ($i = 1$) einmal mehr ausgeführt werden.

5.5.2 Positionsunterschiede

Unterschiede zwischen Prozessinstanzmodellen können sich auch dann ergeben, wenn zwar die Aktivitäten eines Typs y gleichhäufig ausgeführt werden können, jedoch ihre Positionen in der Kontrollflussdefinition derart ist, dass sie nicht simultan ausgeführt werden können.

Das ist für einen Aktivitätstyp y der Fall, wenn

$$\min\left(|w|_{nsf(y,0)}, |w|_{nsf(y,1)}\right) > 0$$

gilt.

Beispiel 5.7 (Positionsunterschiede in $\hat{p}_2$ und $\hat{p}_3$) Im Beispiel von Abbildung 5.5 trifft dies auf die p-Aktivität zu, da

$$\min\left(|w|_{nsf(p,0)}, |w|_{nsf(p,1)}\right) = 1 > 0$$

für

$$w = \text{nsf(p,1)nsf(p,0)} \in \mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\text{pg}'(\hat{p}_2,\hat{p}_3))})$$

gilt.

Hinsichtlich der Positionsunterschiede stellen die Prozessinstanzmodelle $\hat{p}_2$ und $\hat{p}_3$ bereits Spezialfälle dar. Es ist nämlich auch

$$w' = \text{nsf(n,1)nsf(n,0)} \in \mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\text{pg}'(\hat{p}_2,\hat{p}_3))})$$

und

$$\min\left(|w'|_{nsf(n,0)}, |w'|_{nsf(n,1)}\right) = 1 > 0.$$

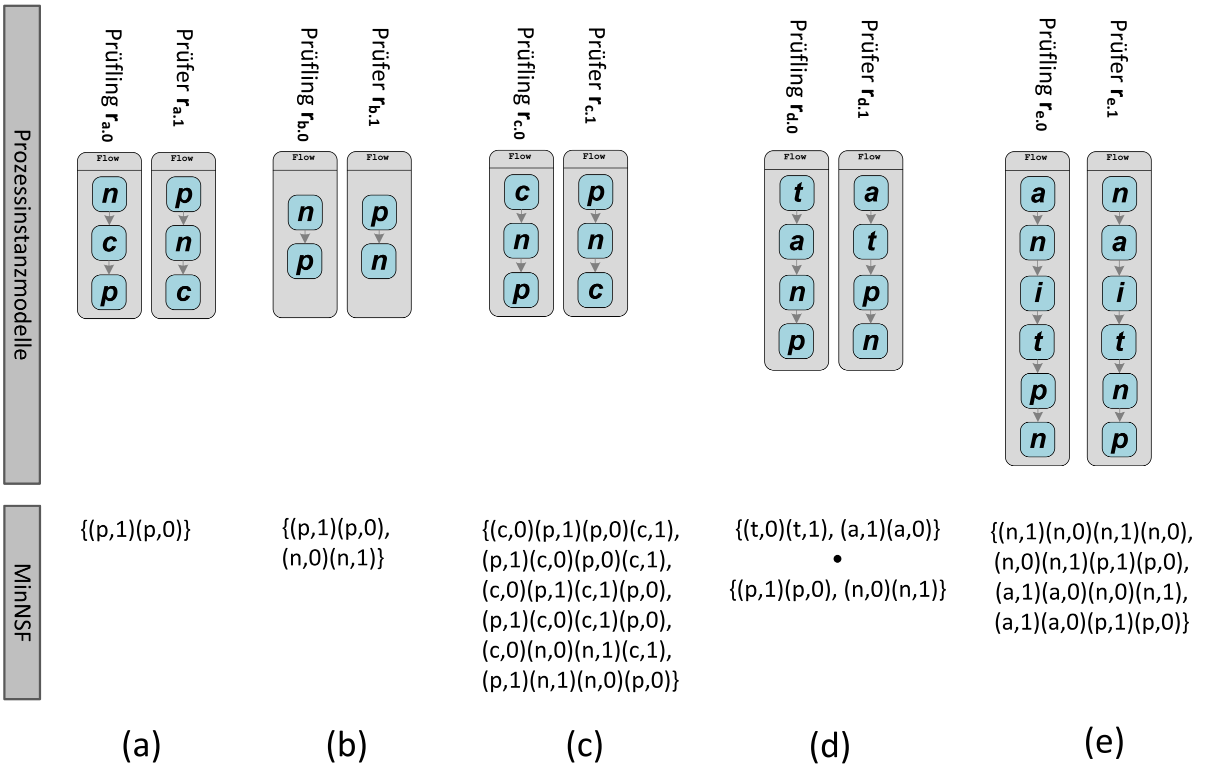


Abbildung 5.8: Spezialfälle von Positionsunterschieden

Insofern kann hier die unterschiedliche Positionierung nicht einer bestimmten Aktivität eines bestimmten Typs zugerechnet werden. Vielmehr sind die *p*- und *n*-Aktivitäten zueinander unterschiedlich positioniert. Das folgende Beispiel diskutiert diesen Sachverhalt detaillierter an weiteren Prozessinstanzmodellen.

Beispiel 5.8 (Spezialfälle bei Positionsunterschieden)
In Abbildung 5.8 sind fünf Paare von Prozessinstanzmodellen abgebildet. Die Kontrolflussdefinition in den Prozessinstanzmodellen ist absichtlich einfach gehalten, d.h. verzichtet auf Verzweigungen, Schleifen und komplizierte Link-Strukturen, damit die Vorkommen von nsf-Transitionen auf den günstigsten Pfaden in den jeweiligen Transitionssystemen einfach nachvollzogen werden können. Die zu den günstigsten Pfaden korrespondierenden kürzesten Wörter, also die jeweilige MinNSF-Sprache, sind in der unteren Hälfte der Abbildung aufgeführt. Hierbei wurde aus den Symbolen der Wörter das wiederkehrende nsf der besseren Lesbarkeit halber ausgelassen. Die MinNSF-Sprachen kommen wie folgt zustande:

(a) Auf dem alleinigen günstigsten Pfad müssen die *p*-Aktivitäten nichtsimultan finalisiert werden. Sollen die *p*-Aktivitäten simultan finalisiert werden, müssen zunächst die n- und dann die c-Aktivität im Prüfling und später im Prüfer nicht-simultan finalisiert werden. Ein solcher Pfad im Transitionssystem wäre kein günstigster. In diesem Spezialfall kann der Positionierungsunterschied den p-Aktivitäten zugerechnet werden.

(b) Bei diesen Prozessinstanzmodellen ergibt sich die gleiche Situation wie bei $\hat{p}_2$ und $\hat{p}_3$. Hier ist allerdings ganz offensichtlich, dass der Positionierungsunterschied nicht einem der beiden Aktivitätstypen zugerechnet werden kann. Für die Kosten des Pfades ist es nämlich unerheblich, ob die n- oder die p-Aktivität nicht-simultan finalisiert wird.

(c) Dieser Fall ist insoweit komplizierter, als dass für jeden Aktivitätstyp c, n und p ein kürzestes Wort existiert, in dem der Aktivitätstyp innerhalb eines nsf-Symbols vorkommt. Entgegen der anfänglichen Intuition ist es nicht so, dass nur die c- und p-Aktivitäten zueinander unterschiedlich positioniert sind, da auch kürzeste Wörter existieren, in denen entweder c oder p nicht vorkommt, also im Transitionssystem simultan finalisiert wird.

(d) Dieser Fall ist gewissermaßen der zweifache Fall (b) nämlich einmal für die Aktivitätstypen a und t und einmal für n und p. Die MinNSF-Sprache lässt sich hier kompakt als Konkatenation zweier Sprachen kürzester Wörter definieren. Im Transitionssystem wird in den günstigsten Pfaden also zuerst entweder die t-Aktivität oder die a-Aktivität nicht-simultan finalisiert und danach entweder die n-Aktivität oder die p-Aktivität.

(e) Dieser Fall unterscheidet sich von den übrigen insoweit, als dass eine Aktivität, nämlich die n-Aktivität, zweifach vorkommt. Sie ist jeweils so positioniert, dass sie in beiden Fällen an Positionsunterschieden beteiligt ist.

Die diskutierten Spezialfälle zeigen, dass es mitunter nicht sinnvoll ist, Positionierungsunterschiede einer bestimmten Aktivität zuzuordnen. Sinnvoll ist vielmehr, Mengen von Aktivitätstypen zu identifizieren, deren Aktivitäten in den Prozessinstanzmodellen unterschiedlich positioniert sind.

Definition 5.7 (Typen unterschiedlich positionierter Aktivitäten)
Für zwei Prozessinstanzmodelle p und q ist die Menge der Typen AT

unterschiedlich positionierter Aktivitäten $\mathcal{U}(p,q)$ wie folgt definiert:

$$\begin{gathered}\mathcal{U}_j(p,q)\\ := \\ \{a \in AP \mid \exists w \in \mathcal{L}_{MinNSF}(\mathfrak{T}_{(\mathcal{R}_{sim},\mathrm{pg}'(p,q))}) \;: \\ \min\Big(|w|_{nsf(a,0)}, |w|_{nsf(a,1)}\Big) = j\}.\end{gathered}$$

Der Parameter $j \in \mathbb{N}_0$ gibt an, wie häufig ein bestimmter Aktivitätstyp an einem Positionierungsunterschied beteiligt ist. Bezogen auf Beispiel 5.8 ist $\mathcal{U}_1(r_{a.0}, r_{a.1}) = \{p\}$, $\mathcal{U}_1(r_{b.0}, r_{b.1}) = \{n, p\}$ und $\mathcal{U}_1(r_{c.0}, r_{c.1}) = \{n, p, c\}$ sowie $\mathcal{U}_1(r_{d.0}, r_{d.1}) = \{n, p, t, a\}$. Die Mengen $\mathcal{U}_j$ für $j \geq 2$ sind in diesen Fällen jeweils leer. Demgegenüber ist $\mathcal{U}_1(r_{e.0}, r_{e.1}) = \{n, a, p\}$ und $\mathcal{U}_2(r_{e.0}, r_{e.1}) = \{n\}$. Letztere Menge gibt darüber Aufschluss, dass die n-Aktivität zweifach innerhalb unterschiedlich positionierter Aktivitäten auftaucht. Die Menge

$$\mathcal{U}(p,q) := \bigcup_{j>0} \mathcal{U}_j(p,q)$$

abstrahiert von dieser durch j indizierten Feinunterscheidung.

$\mathcal{U}(p,q)$ beinhaltet also alle Aktivitätstypnamen, die als erster Parameter innerhalb von Symbolen kürzester Wörter in der MinNSF-Sprache einmal mit 0 und einmal mit 1 als zweiten Parameter vorkommen.

5.6 Konsistenzbericht

Die Kosten der günstigsten Pfade und insbesondere die Ausführungshäufigkeiten und Positionsunterschiede werden Prozessbeteiligten in einem tabellarischen Konsistenzbericht präsentiert. Dessen konkrete Form und die Bedeutung seiner Inhalte werden im Folgenden erläutert.

5.6.1 Implementierung

Abbildung 5.9 zeigt einen Screenshot nach Durchführung einer Konsistenzprüfung. Die Wissensbasis ist der Nachvollziehbarkeit halber sehr klein gehalten und beschränkt sich auf die Prozessmodelle p_1 bis p_3, die am Kapitelanfang anhand von Abbildung 5.2 diskutiert wurden. Zusätzlich ist ein Prozessinstanzmodell p_2' enthalten, das sich von p_2 nur darin unterscheidet, dass die

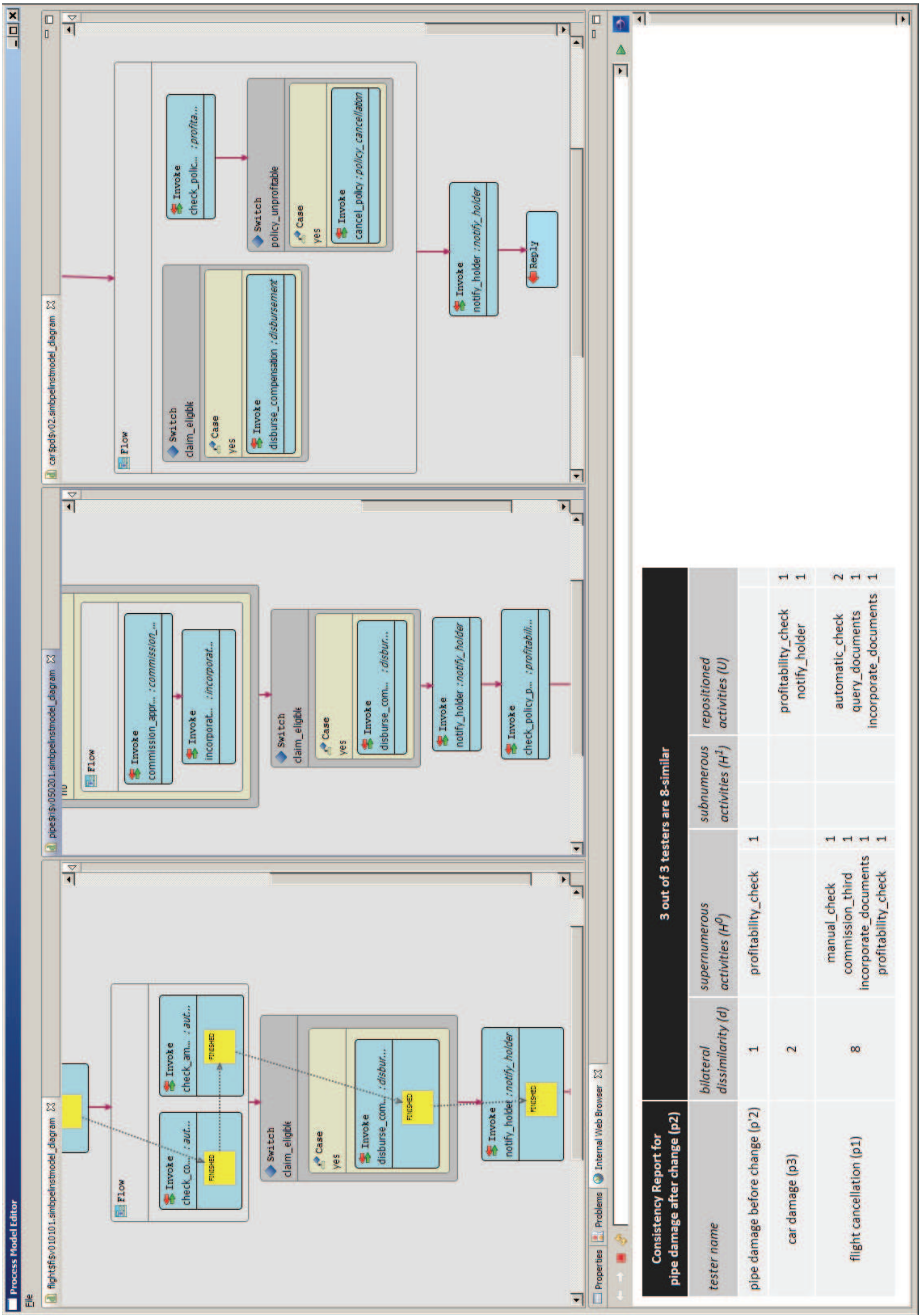


Abbildung 5.9: Prozessmodelleditor mit Konsistenzbericht

profitability_check-Aktivität in p_2' fehlt. Das Prozessinstanzmodell p_2 geht also aus p_2' durch dynamisches Einfügen der *profitability_check*-Aktivität hervor.

Die Prozessdefinitions- und -instanzmodelle p_1 bis p_3 sind – aus Platzgründen nur teilweise – im oberen Bereich des Screenshots dargestellt. Im unteren Bereich ist der Konsistenzbericht angezeigt, der Ergebnis der Konsistenzprüfung des Prüflings p_2 (mittleres Prozessmodell) ist.

Die schwarz hinterlegte Kopfzeile der Tabelle gibt eine grobe Auskunft über die Ähnlichkeit des Prüflings gegenüber der gesamten Wissensbasis M_{kb}. Im konkreten Fall sind alle drei Prüfer (3 out of 3 testers) 8-ähnlich zur p_2 (vgl. Definition 5.3). Der Schwellenwert t, in diesem Fall 8, lässt sich dabei individuell für jedes Prozessmodell als Prüfling konfigurieren.

In den folgenden Zeilen werden die Informationen aufgelistet, die die paarweise Detailanalyse des Prüflings und je einem 8-ähnlichen Prüfer ergab. Jede Zeile beginnt mit dem Namen des betreffenden Prüfers (tester name) gefolgt von der jeweiligen bilateralen Prozessunähnlichkeit, die einen ersten quantitativen Aufschluss über die Ähnlichkeit von Prüfling und Prüfer liefert (vgl. Definition 5.2).

Darauf folgen die Detailangaben über Häufigkeitsunterschiede, getrennt nach überzähligen (supernumerous) und unterzähligen (subnumerous) Aktivitäten mit Bezug auf den Prüfer. Diese entsprechen den Mengen $\mathcal{H}^0$ bzw. $\mathcal{H}^1$, wie sie in Unterabschnitt 5.5.2 definiert wurden. Die Angaben sind ergänzt um Zahlen k, die die Zugehörigkeit zur $\mathcal{H}^i$-Partition $\mathcal{H}^i_k$ anzeigen (vgl. Definition 5.6).

Es kann dem Konsistenzbericht entnommen werden, dass eine *profitability_check*-Aktivität tatsächlich einmal häufiger im geänderten Prozessinstanzmodell p_2 gegenüber dem ursprünglichen Prozessinstanzmodell p_2' ausgeführt wird, was gerade durch die dynamische Einfügung erzielt werden sollte.

Ganz rechts im Konsistenzbericht sind die Positionsunterschiede dargestellt. Wie zu erwarten sind die *profitability_check*- und die *notify_holder*-Aktivität in p_2 und p_3 unterschiedlich positioniert. Die Angabe entsprechen der Menge $\mathcal{U}$ aus Definition 5.7. Wiederum gibt die rechts neben den Aktivitätstypen stehende Zahl die Zuordnung zur $\mathcal{U}$-Partition $\mathcal{U}_i$ an.

Beispielsweise sind in p_1 beide *automatic_check*-Aktivitäten unvereinbar mit p_2 positioniert, daher die Zahl 2 rechts neben automatic_check im Konsistenzbericht.

5.6.2 Diskussion

Der Konsistenzbericht, der im Wesentlichen eine konkrete Darstellung der Definitionen der vorherigen Abschnitte darstellt, ist dem Prozessbeteiligten dann von Nutzen, wenn er die berechneten Mengen aus den Definitionen der vorhergehenden Abschnitte richtig interpretiert.

Es ist möglich, aufbauend auf den berechneten Mengen, Antworten auf typische Fragestellungen des Prozessbeteiligten abzulesen, die die Konsistenz seiner dynamischen Änderungen in einem Prozessinstanzmodell zu anderen Prozessinstanzmodellen betreffen. Die Fragestellungen betreffen insbesondere die Wirksamkeit und Richtigkeit von dynamischen Änderungen in Prozessinstanzmodellen und sind verwandt mit den Fragestellungen, die auch durch Komplianzprüfungen gegen explizites Prozesswissen beantwortet werden können (sofern explizites Prozesswissen vorliegt). Der für den Prozessbeteiligten erzeugte Konsistenzbericht enthält neben den berechneten Mengen Antworten auf diese Fragestellungen, die im Folgenden erläutert werden.

Wirksamkeit dynamischer Änderungen

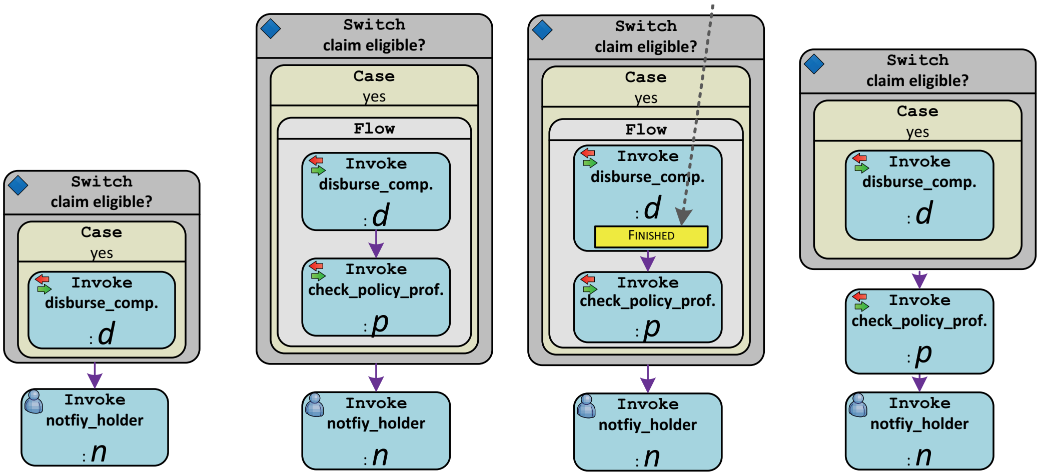
Für einen Prozessbeteiligten stellt sich im Zuge einer dynamischen Änderung die Frage nach ihrer Wirksamkeit. Es ist gerade bei umfangreichen Prozessen sehr leicht möglich, eine dynamisch eingefügte *x*-Aktivität unabsichtlich so zu platzieren, dass sie nicht oder nur unter gewissen Umständen ausgeführt wird.

Es ist nun im Prozessmodelleditor möglich, jeweils ein lokales Prozessinstanzmodell separat vor und nach einer dynamischen Änderung als unterschiedliche Modelle p bzw. q zu speichern. Die Berücksichtigung von p als Prüfer für q kann zudem durch Konfigurierung des Prozessmodelleditors unabhängig von der Höhe von $d(p,q)$ erzwungen werden. Es gilt dann Folgendes: Eine dynamisch eingefügte *x*-Aktivität wird tatsächlich zusätzlich ausgeführt, wenn $\mathcal{H}^1(p,q) = \{x\}$ gilt.

Im Beispiel 5.9 wird dies an verschiedenen Prozessmodell(-fragmenten) vertieft diskutiert.

Beispiel 5.9 (Dynamisch eingefügte Aktivität) Sei $\hat{r}_{2.0}$ wie in Abbildung 5.10(a), also gleich dem Prozessinstanzmodell(-fragment) $\hat{p}_2$ aus Abbildung 5.2 nur ohne *profitability_check*-Aktivität, im Folgenden wieder abgekürzt als *p*-Aktivität bezeichnet. Durch dynamische Änderungen im Prozessinstanzmodell können nun beispielsweise folgende Varianten – abgebildet durch $\hat{r}_{2.1}$ bis $\hat{r}_{2.3}$ – zustande kommen:

1. In $\hat{r}_{2.1}$ befindet sich noch keine Zustandsinformation. Demzufolge ist noch nicht determiniert, ob der Zweig (`Case`-Aktivität) durchgeführt wird. Wird nun die *p*-Aktivität hinter der d-Aktivität eingefügt, so ist nicht gesichert, dass diese Aktivität tatsächlich auch ausgeführt wird.



(a) $\hat{r}_{2.0}$ (b) $\hat{r}_{2.1}$ (c) $\hat{r}_{2.2}$ (d) $\hat{r}_{2.3}$

Abbildung 5.10: Wirksamkeit dynamischer Änderungen

Es gilt daher

$$\mathcal{H}^1(\hat{r}_{2.0}, \hat{r}_{2.1}) = \emptyset.$$

Aus dieser Information im Konsistenzbericht kann der Prozessbeteiligte entnehmen, dass seine dynamischen Änderungen unter Umständen unwirksam ist. Dies ist beispielsweise dann ungewollt, wenn in der frühen Phase der Schadensbearbeitung bemerkt wird, dass der Kunde auffällig häufig Ersatzansprüche geltend macht und daher die Vertragsüberprüfung (check_policy_profitability) unabhängig davon geschehen soll, ob die Auszahlung im gerade laufenden Prozess durchgeführt werden soll oder nicht.

2. In $\hat{r}_{2.2}$ ist der Kontrollfluss bereits bis zur Aktivität disburse_compensation fortgeschritten. Daher ist determiniert, dass die Case-Aktivität durchgeführt wird. Folglich wird auch die zur Case-Aktivität dynamisch hinzugefügte p-Aktivität check_policy_profitability ausgeführt. Dementsprechend gilt

$$\mathcal{H}^1(\hat{r}_{2.0}, \hat{r}_{2.2}) = \{p\},$$

woraus der Prozessbeteiligte entnehmen kann, dass die p-Aktivität tatsächlich durchgeführt wird.

3. Ähnlich zum zweiten Fall verhält es sich mit $\hat{r}_{2.3}$, obwohl sich $\hat{r}_{2.2}$ und $\hat{r}_{2.3}$ offenkundig syntaktisch deutlich voneinander unterscheiden. In $\hat{r}_{2.3}$ ist wiederum wie in $\hat{r}_{2.1}$ der Kontrollfluss noch nicht bis in das

Fragment vorangeschritten. Im Unterschied zu $\hat{r}_{2.1}$ ist die p-Aktivität jedoch außerhalb der Case-Aktivität eingefügt worden. Folglich wird sie unabhängig von der Kontrollflussposition auf jedem Ausführungspfad durchgeführt. Wiederum gilt wie im zweiten Fall

$$\mathcal{H}^1(\hat{r}_{2.0}, \hat{r}_{2.3}) = \{p\}.$$

Durch diese Änderung hat der Prozessbeteiligte bewirkt, dass die Vertragsprüfung auf jeden Fall stattfindet und nicht nur dann, wenn auch eine Auszahlung im konkreten Fall geschieht. Die Wirksamkeit seiner Änderung kann er an $\mathcal{H}^1(\hat{r}_{2.0}, \hat{r}_{2.3})$ ablesen.

Die Wirksamkeit dynamischer Löschungen oder Wiederholungen kann auf ähnliche Weise ermittelt werden. Bei dynamischen Löschungen kann abgelesen werden, ob die gelöschte Aktivität im ursprünglichen Modell zusätzlich durchgeführt wird, bei dynamischen Wiederholungen, ob die wiederholten Aktivitäten tatsächlich durchgeführt werden.

Die Überprüfung der Wirksamkeit dynamischer Änderungen bezweckt Ähnliches wie die Auswertung von includes-Bedingungsbeziehungen in expliziten Prozesswissensmodellen (vgl. Unterabschnitt 2.2.3). Eine fehlerhafte Ausführungshäufigkeit einer bestimmten Aktivität, einschließlich ihrer gänzlichen Abwesenheit in einem Prozessinstanzmodell, kann also auf zwei Weisen festgestellt werden: (1) Ist eine entsprechende includes-Bedingungsbeziehung modelliert, explizites Prozesswissen also vorhanden, kann eine eventuelle Komplianzverletzung in einem Prüfling p, wie in Kapitel 4 beschrieben, ermittelt werden. (2) Ist das Prozesswissen unvollständig, fehlt also eine entsprechende includes-Bedingungsbeziehung, so besteht zusätzlich die Möglichkeit zu Konsistenzprüfungen gegen implizites Prozesswissen. In Analogie zu includes-Bedingungsbeziehungen können über die zuvor erläuterten Mengen Unterschiede der Ausführungshäufigkeiten relativ zu hinreichend ähnlichen Modellen q ermittelt werden.

Abfolgerichtigkeit dynamischer Änderungen

Neben der reinen Wirksamkeit dynamischer Änderungen ist für den Prozessbeteiligten auch die Richtigkeit seiner Änderungen bzgl. der Abfolge einer dynamisch eingefügten Aktivität interessant. Beispielsweise ist das dynamische Einfügen der p-Aktivität check_policy_profitability *nach* der n-Aktivität notify_holder nicht sinnvoll, da im Zuge der p-Aktivität für den Versicherungsnehmer natürlich Beschlüsse gefasst werden, die dem Versicherungsnehmer unbedingt mitgeteilt werden müssen.

Anhand von $\mathcal{U}$ lassen sich bezogen auf eine dynamisch eingefügte *p*-Aktivität in einen Prüfling r gegenüber einem Prüfer q folgende Schlüsse ziehen:

1. Die Aktivität ist problematisch positioniert, wenn $p \in \mathcal{U}(r,q)$ gilt. Diese Enthaltenseinsbeziehung gibt allerdings nur einen groben Hinweis auf die Problematik. Wie in Beispiel 5.8 dargelegt wurde, kann der Positionierungsfehler nur in Ausnahmefällen einer einzelnen Aktivität eines bestimmten Typs zugeordnet werden.

2. Der besagte Ausnahmefall ist gegeben, wenn $\mathcal{U}_1(r,q) = \{p\}$ gilt. Hieran kann der Prozessbeteiligte ablesen, dass eindeutig die *p*-Aktivität in der Kontrollflussdefinition neu positioniert werden muss, um die Positionierungsunterschiede zwischen Prüfling r und Prüfer q zu nivellieren.

Beispiel 5.10 (Fehlplatzierte *p*-Aktivität)
Angenommen, in Abbildung 5.2 wurde im Prozessinstanzmodell p_2 die *p*-Aktivität dynamisch eingefügt. Dann liefert der Vergleich mit p_3 als Prüfer aus derselben Abbildung:

$$p \in \mathcal{U}(p_2, p_3) = \mathcal{U}_1(p_2, p_3) = \{\mathsf{n}, \mathsf{p}\}.$$

Der oben beschriebene zweite Fall kommt hier nicht zur Geltung, da weder der *n*- noch der *p*-Aktivität der Positionierungsfehler zugeordnet werden kann, sondern nur die Aussage getroffen werden kann, dass beide Aktivitäten zueinander in beiden Prozessinstanzmodellen unterschiedlich positioniert sind.

Wie bei der Wirksamkeit dynamischer Änderungen aus Unterabschnitt 5.6.2 gibt es auch bei der Abfolgerichtigkeit Analogien zu den Bedingungsbeziehungen expliziten Prozesswissens, naheliegenderweise zu den abfolgeorientierten Bedingungsbeziehungstypen succeeded_by und preceded_by. Im Fall von Prozessinstanzmodell p_2 wäre im expliziten Prozesswissen eine Bedingungsbeziehung $p \xrightarrow[1..*]{succeeded\_by} n$ erforderlich, damit auf Basis von Komplianzprüfungen auf die problematische Positionierung der *p*-Aktivität hingewiesen werden kann.

5.7 Diskussion

Der in diesem Kapitel vorgestellte Ansatz zur Konsistenzprüfung von Prozessinstanzmodellen, über die in den Prozessinstanzmodellen implizit vorhandenes Prozesswissen ausgenutzt wird, hat bestimmte Stärken und Schwächen, die in diesem Abschnitt erläutert werden, bevor sie im nächsten Abschnitt zum Vergleich dieses Ansatzes mit verwandten Arbeiten wiederaufgenommen werden.

5.7.1 Stärken

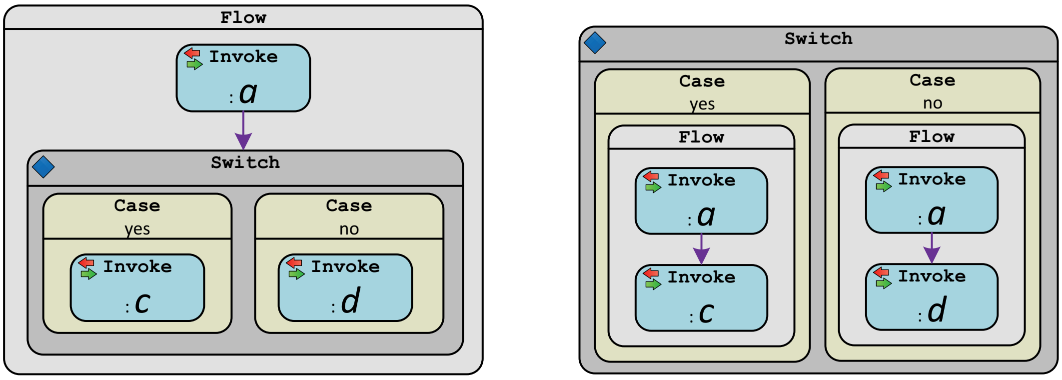
Einige Stärken des Ansatzes wurden eingangs bereits angedeutet und werden an dieser Stelle vertieft.

Abstraktion von der Modellsyntax

Eine Stärke des Ansatzes ist die Tatsache, dass die Ähnlichkeitsberechnungen zwischen Prozessinstanzmodellen nicht auf Basis der Prozessmodellsyntax durchgeführt werden, sondern stattdessen die Abläufe, die die syntaktischen Konstrukte der Prozessinstanzmodelle modellieren, miteinander verglichen werden. Hierdurch ergeben sich Vorteile:

Übertragbarkeit Der Ansatz kann so modifiziert werden, dass er auf andere Prozessmodellierungssprachen und Ähnlichkeitsbegriffe passt.

Andere Prozessmodellierungssprachen Die Graphersetzungsregeln in der Regelmenge $\mathcal{R}_{sim}$ bilden die Prozessinstanzmodelle auf ein Graphtransitionssystem ab, das alleiniger Gegenstand weiterer Analysen ist. Die Abstraktion von der konkreten Modellierungssprache, in diesem Fall SimBPEL-Instance, geschieht also in einer frühen Phase und beeinflusst spätere Schritte nicht. Änderungen und insbesondere Erweiterungen in der Modellierungssprache können allein durch Anpassungen der Graphersetzungsregeln in $\mathcal{R}_{sim}$ beantwortet werden. Da sich gängige Prozessmodellierungssprachen wie WS-BPEL, BPMN oder XPDL in der Menge und Semantik verschiedener syntaktischer Konstrukte ähneln, ist der Ansatz auch auf diese ähnlichen Sprachen übertragbar. Vorhandene Graphersetzungsregeln in $\mathcal{R}_{sim}$ müssen dabei ggf. auf die syntaktischen Unterschiede angepasst werden. Beispielsweise werden in BPMN bei Verzweigungen verschiedene Zweige statt durch Schachtelung in einem `Switch`-Element durch Kanten modelliert, die aus einem `Switch`-ähnlichen Element auslaufen.



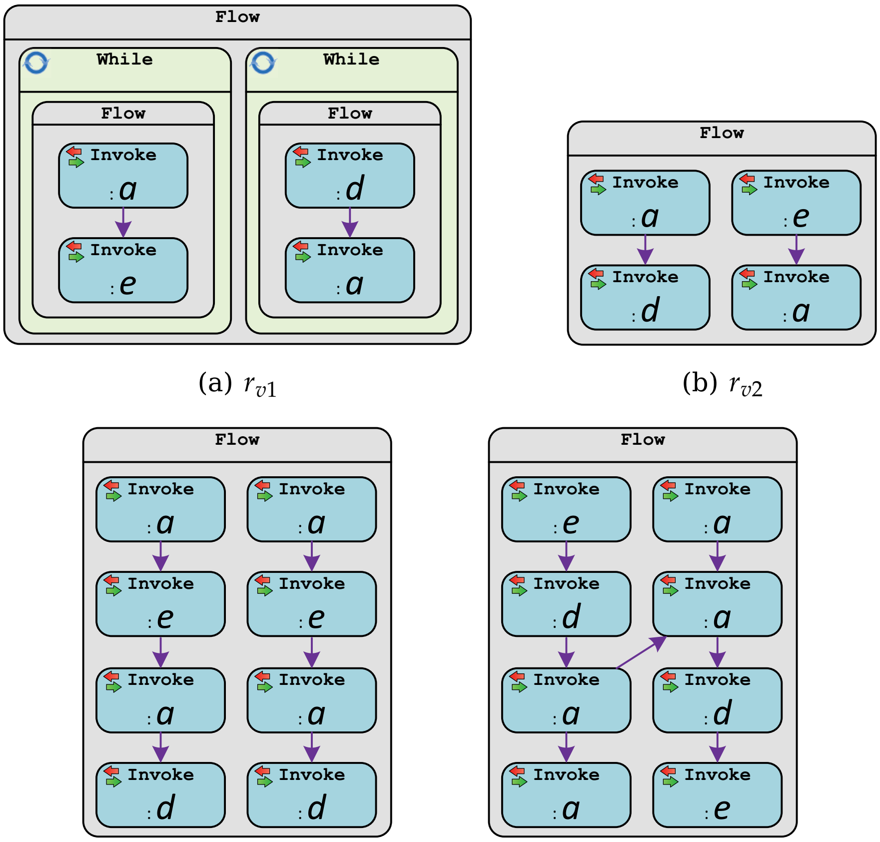
(a) r_{bi1} (b) r_{bi2}

Abbildung 5.11: Nicht-bisimilare Prozessinstanzmodelle mit identischer Prozesslaufmenge

Andere Ähnlichkeitsbegriffe Die Anwendungen der Regeln in $\mathcal{R}_{sim}$ werden durch Priorisierung so gesteuert, dass nach Möglichkeit nur die günstigsten Pfade im resultierenden Transitionssystem enthalten sind, da nur diese für die Berechnung t-ähnlicher Prozessinstanzmodelle und die Detailanalyse relevant sind. Es ist jedoch auch möglich, die Regeln und die Priorisierung so abzuändern, dass die Ähnlichkeitsberechnung anderen Ähnlichkeitsbegriffen folgt.

Beispiel 5.11 (Bisimilarität zwischen Prozessinstanzmodellen) In Abbildung 5.11 sind zwei Prozessinstanzmodelle dargestellt, die offensichtlich syntaktisch verschieden sind, aber die gleichen Prozessläufe besitzen: $c(r_{bi1}) = c(r_{bi2}) = \{ac, ad\}$. Insofern ist die bilaterale Prozessunähnlichkeit $d(r_{bi1}, r_{bi2}) = 0$. Die Prozessinstanzmodelle sind jedoch nicht bisimilar, wobei sich die Bisimilarität auf die Zustandstupel der getypten `Invoke`-Aktivitäten bezieht. Die Bisimilarität kann in einem einfachen Simulationsspiel wiederlegt werden, in dem der Falsifizierer im zweiten Zug die c- bzw. d-Aktivität zur Durchführung wählt, wenn der Verifizierer zuvor die a-Aktivität gewählt hat, die im gleichen Zweig (`Case`-Aktivität) ist, wie die d- bzw. c-Aktivität.

Zur Feststellung, ob ein Prozessinstanzmodell p similar zu q ist, kann die Regelmenge $\mathcal{R}_{sim}$ so abgeändert werden, dass zunächst nur Regelanwendungen im zu q gehörigen Teil des dualen Prozessgraphen erlaubt sind (Falsifiziererzug), bis eine `Invoke`-Aktivität finalisiert wurde und danach nur Regelanwendungen durchgeführt werden, die den zu p gehörigen Teil ändern



(a) r_{v1} (b) r_{v2}

(c) r_{v3} (d) r_{v4}

Abbildung 5.12: Nicht-offensichtliche gemeinsame Prozessläufe

(Verifizierererzug), bis dort eine gleichgetypte Aktivität finalisiert wurde, wonach wieder der Falsifizierer am Zug ist. Dann ist p similar zu q, wenn von jeden Zustand aus, der durch einen Falsifizierererzug erreicht wurde, eine Endzustand erreichbar ist, also ein Zustand, in dem alle Aktivitäten entweder im Zustand Finished oder Dead sind. Das beschriebene Vorgehen wurde anhand einer Regelmengenvariante $\mathcal{R}_{bisim}$ von $\mathcal{R}_{sim}$ überprüft. $\mathcal{R}_{bisim}$ hat allerdings aufgrund des zu strengen Ähnlichkeitsbegriffs für das Gesamtvorhaben der Arbeit keine weitere Bedeutung.

Vollständigkeit Unabhängig davon, wie sehr sich zwei Prozessinstanzmodelle in ihrer Syntax unterscheiden, d.h. in ihrer Anzahl von Aktivitäten und ihrer Anordnung in der Kontrollflussdefinition, kann durch das in Abschnitt 5.4 beschriebene Verfahren die bilaterale Prozessunähnlichkeit zweier Prozessinstanzmodelle bestimmt werden. Die bisherigen, am Anwendungsbeispiel angelehnten Beispiele verdeutlichen diese Stärke nicht.

Beispiel 5.12 (Nicht-offensichtliche gemeinsame Prozessläufe)
Abbildung 5.12 zeigt einige Prozessinstanzmodelle. Der Vergleich von r_{v1} mit den anderen Prozessinstanzmodellen der Abbildung ergibt Folgendes:

- Es ist $c(r_{v1}, r_{v2}) \neq \emptyset$, obwohl die `Links` in den Modellen unterschiedliche Aktivitäten verbinden.
- Es ist $c(r_{v1}, r_{v3}) = \emptyset$ und $d(r_{v1}, r_{v3}) = 2$. Es kann in diesem Spezialfall kein gemeinsamer Prozesslauf existieren, da r_{v3} die Ausführung von *d*-Aktivitäten erfordert und mindestens eine davon als letzte im Prozess durchgeführt werden muss, laut r_{v1} aber eine *d*-Aktivität nicht zuletzt durchgeführt werden kann.
- Demgegenüber besitzen r_{v1} und r_{v4} einen gemeinsamen Prozesslauf ($c(r_{v1}, r_{v4}) = \{\mathsf{aedaadae}\}$), was aufgrund der syntaktischen Unterschiede der beiden Prozessinstanzmodelle nicht offensichtlich ist.

Historienberücksichtigung

In Abschnitt 2.5 wurde erläutert, dass die Semantik von Prozessinstanzmodellen unter mehreren Blickwinkeln betrachtet werden kann. Neben der Zukunftssemantik, die sich auf die (noch) möglichen zukünftigen Teilprozessläufe bezieht, hat ein Prozessinstanzmodell eine Historiensemantik, welche auf den bereits determinierten Teil des Prozesslaufs Bezug nimmt. Die Prozesshistorie von laufenden und abgeschlossenen Prozessinstanzmodellen wird durch $\mathcal{R}_{sim}$ berücksichtigt. Dadurch kann ausgeschlossen werden, dass für einen Prüfling p und einen Prüfer q ein günstigster Pfad gefunden wird, der einen pseudo-simultanen Prozesslauf induziert, der jedoch der Historie von p oder q widerspricht. Würde dieser Widerspruch in Kauf genommen, so fänden sich für p zusätzliche, vermeintliche t-ähnliche Prüfer wie q. Die Detailanalyse zwischen p und einem q brächte folglich ebenso widersprüchliche Ergebnisse und trüge zur Verwirrung des Prozessbeteiligten bei.

5.7.2 Lösungen für problematische Aspekte

Bestimmte Qualitätsfaktoren stehen bei dem zuvor vorgestellten transitionsbasierten Ansatz zueinander im Widerstreit. Diese Schwäche wird im Folgenden erläutert und es wird erklärt, wie ihr begegnet werden kann.

Die simulierten Ausführungszustände der Prozessinstanzmodelle in einem Prozessgraphen sind zahlreich. Das trifft umso mehr auf einen dualen Prozessgraphen zu, in dem die Ausführungszustände gleich zweier Prozessinstanzmodelle enthalten sind. Dementsprechend groß werden die erzeugten Transitionssysteme, was sowohl die Speicherplatz- als auch die Laufzeitkomplexität des Gesamtalgorithmus bei der Erzeugung der Transitionssysteme als auch bei der Ermittlung günstigster Pfade erhöht. Es gibt eine Reihe von Möglichkeiten, die Größe erzeugter Transitionssysteme auch für größere duale Prozessgraphen zu reduzieren. Die zentrale Herausforderung dabei ist, die Erzeugung nicht-günstigster Pfade in den Transitionssystemen zu unterbinden, günstigste Pfade aber vollständig zu erhalten.

Syntaktische Teilidentitäten

Eine Grundannahme bei der Konsistenz- (wie auch bei der Komplianzprüfung) ist, dass gleichgetypte Aktivitäten fachlich Gleiches bedeuten. Es ist jedoch nicht in jedem Fall nötig, die Gleichheit der Abläufe zweier Prozessinstanzmodelle dadurch zu ermitteln, indem alle gleichgetypten atomaren `Invoke`-Aktivitäten schrittweise simultan finalisiert werden. Mitunter finden sich in verglichenen Prozessinstanzmodellen auch identische komplexe Aktivitäten, also solche Aktivitäten, deren Kindaktivitäten wiederum identisch sind und deren zugehörige Kontrollflussdefinition ebenfalls gleich ist.

Syntaktisch identische komplexe Aktivitäten werden bei der pseudo-simultanen Durchführung zweier Prozesse während der Vorbereitungsphase als Paare markiert. Dabei werden zwei komplexe Aktivitäten als syntaktisch gleich erkannt, wenn alle ihre direkten Kinder als syntaktisch gleich erkannt wurden und die Kontrollflussdefinition zwischen den direkten Kindern identisch ist. Beispielsweise beim Vergleich von p_2 und p_3 aus Abbildung 5.2 kann das oberste komplexe `Flow`-Element simultan in einem Schritt finalisiert werden.

Regelpriorisierung

Die Anwendung der Regeln in $\mathcal{R}_{sim}$ wird durch einen Automaten gesteuert. Über diesen Automaten kann erst erreicht werden, dass zunächst eine Vorbereitung des dualen Prozessgraphen stattfindet, bevor die pseudo-simultane Ausführungsphase beginnt. Des Weiteren können die zur Ausführungsphase gehörigen Regeln so priorisiert werden, dass einerseits günstigste Pfade erhalten bleiben, andererseits das Transitionssystem aber um viele Zustände reduziert werden kann. Die Regeln sind wie folgt priorisiert, beginnend mit der höchsten Priorität:

1. Finalisierung von (komplexen) Aktivitäten (Regel finCplx) bzw. gleichberechtigt Zurücksetzen von `While`-Aktivitäten zur erneuten Iteration (resetWhile).

2. Simultane Finalisierung von Aktivitäten (sf).

3. Aktivierung von Aktivitäten (act) und gleichberechtigt Deaktivierung von optional durchgeführten Aktivitäten (markDead).

4. Nicht-simultane Finalisierung von Aktivitäten (nsf).

5. Prüfung auf das Endzustandskriterium (completed).

5.7.3 Mögliche Erweiterungen

Der Ansatz könnte durch noch weitere Optimierungen verbessert werden, die im Rahmen dieser Arbeit allerdings nicht umgesetzt wurden.

Verbindung mit partiellem explizitem Prozesswissen

Das in den vorherigen Abschnitten erläuterte Verfahren zur Nutzbarmachung von implizitem Prozesswissen durch Konsistenzchecks zwischen Prozessinstanzmodellen bietet Raum für weitere Optimierungen. Wie eingangs in diesem Kapitel erwähnt, ist explizites Prozesswissen (in Prozesswissensmodellen) häufig unvollständig. Insbesondere die Modellierung von Bedingungsbeziehungen ist aufwändig, da jede Bedingungsbeziehung dahingehend kritisch überprüft werden muss, ob sie gewollte Spezialfälle ggf. ausschließt bzw. ungewollte einschließt, so dass sie umformuliert werden muss.

Umgekehrt ist die Modellierung von Spezialisierungsbeziehungen zwischen Aktivitätstypen in Prozesswissensmodellen vergleichsweise unaufwändig, da in vielen Anwendungsdomänen eine Klassifikation von Prozessen, beispielsweise über Prozesskataloge, ohnehin vorliegt. Dieses partielle Prozesswissen kann zur Optimierung von Konsistenzcheck nach dem zuvor in diesem Kapitel beschriebenen Verfahren nutzbar gemacht werden.

Spezialisierungsbeziehungen in Prozesswissensmodellen können zur Einschränkung der Grundmenge von Prüfern bei der Berechnung zum Prüfling p t-ähnlicher Prozessinstanzmodelle verwendet werden. Ausgangspunkt ist die Annahme, dass der Typ sehr unähnlicher Prozessinstanzmodelle innerhalb eines Anwendungsbereichs, beispielsweise die Bearbeitung eines Leitungswasserschadens und die Policierung einer Lebensversicherung, in der Aktivitätstyphierarchie bzgl. der Vererbungsbeziehungen weit voneinander entfernt sind bzw. überhaupt keinen gemeinsamen Vorfahren haben.

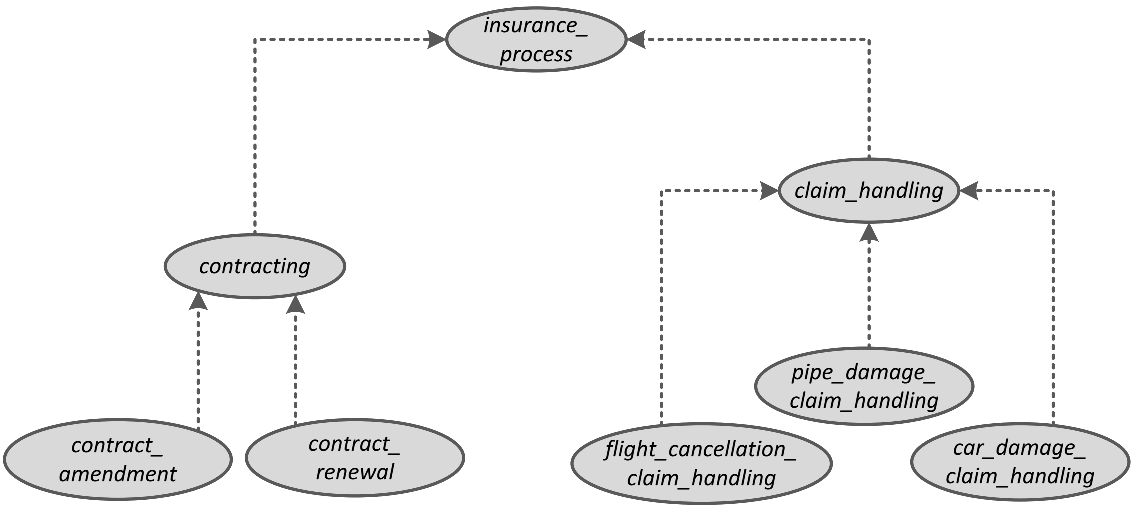


Abbildung 5.13: Aktivitätstyphierarchie für Prozessinstanzmodelle

Definition 5.8 (Aktivitätstypabstand) Sei (V_B, E_B) ein als Graph aufgefasstes Prozesswissensmodell in BPCL, wobei V_B die Aktivitätstypen und E_B die Verbungsbeziehungen sind. Sei ferner

$$path \;:\; V_B \times V_B \to \mathbb{N}_0^\infty;\; (x,y) \mapsto \begin{cases} x = y & 0 \\ (x,y) \in E_B & 1 \\ (x,y) \notin E_B & \min_{(x,z)\in E_B}(1 + path(z,y)) \end{cases}$$

die Länge des kürzesten Pfades von x nach y. Dann ist

$$ad \;:\; V_B \times V_B \to \mathbb{N}_0^\infty;\; (x,y) \mapsto \min_{a\in V_B}(path(x,a) + path(y,a))$$

eine Abbildung, die zwei Aktivitätstypen einen (ggf. unendlichen) Abstand zuordnet und

$$\begin{aligned} M_{ad} \;:\; & \mathbb{N}_0 \times M_{kb} \to V_B; \\ & (n,p) \mapsto \{q \in M_{kb} |\; p \text{ bzw. } q \text{ hat Typ } v_p \text{ bzw. } v_q \text{ und } ad(v_p, v_q) \leq n\} \end{aligned}$$

die Menge der Prozessinstanzmodelle q, deren Typ v_q von einem Typ v_p eines Prozessinstanzmodells p weniger als n entfernt sind.

Beispiel 5.13 (Aktivitätstyphierarchie für Prozessinstanzmodelle)
In Abbildung 5.13 ist eine einfache Vererbungshierarchie von Aktivitätstypen für Prozessinstanzmodelle dargestellt. Angenommen ein Prüfling p modelliert die Bearbeitung eines Leitungswasserschadens, hat also den Typ *pipe_damage_claim_handling*. Dann ist

$$\begin{aligned} M_{ad}(2,p) = \{q \in M_{kb} \mid q \text{ ist vom Typ} \\ &\textit{pipe\_damage\_claim\_handling} \text{ oder} \\ &\textit{flight\_cancellation\_claim\_handling} \text{ oder} \\ &\textit{car\_damage\_claim\_handling} \text{ oder } \textit{claim\_handling}\} \end{aligned}$$

Alle übrigen Prozessinstanzmodelle, die beispielsweise Vertragsabschlüsse modellieren (*contracting*), sind in der Menge nicht enthalten, da sie im Prozesswissensmodell weiter als 1 von p entfernt sind.

Nutzbringend ist die Analyse der Aktivitätstypabstände bei der Berechnung t-ähnlicher Prozessinstanzmodelle. Es können im Vorhinein die Kandidaten t-ähnlicher Prüfer für p auf eine Menge $M_{ad}(n,p)$ eingeschränkt werden, d.h. in Abwandlung von Definition 5.3

$$M_{(t,n)}(p) := \{q \in M_{ad}(n,p) \mid d(p,q) \leq t\}$$

angesetzt werden. Hierdurch wird die zeit- und speicherplatzkomplexe, graphtransitionssystembasierte Analyse von potentiellen Prüfern

$$q' \in M_t(p) - M_{(t,n)}(p)$$

vermieden, von denen ohnehin eine große Unähnlichkeit zu p zu erwarten ist.

Effizienz: Eingriff in die Transitionssystemgenerierung

Es wäre sinnvoll, die Generierung von Transitionssystemen durch GROOVE so zu beeinflussen, dass von einem Zustand s_i kein Nachfolger generiert wird, wenn dieser nur über einen bzgl. nsf-Transitionen teureren Pfad vom Startzustand s_1 erreichbar ist, als der bisher günstigste Pfade von s_1 zu einem Endzustand $s_{completed}$. Eben dieses Kriterium wird bei der Suche nach günstigen Pfaden im Transitionssystem angewendet. Zustände, die nur über nicht-günstigste Pfade erreichbar sind, sind ohnehin obsolet. Diese Optimierung erfordert einen Eingriff in die Implementierung von GROOVE und wurde bisher nicht realisiert.

	Wörzberger	Küster et al.	Li et al.	Aalst et al.	Wombacher	Dijkman
Verfahren						
Verfahren	semantikbasiert	syntaxbasiert	syntaxbasiert	semantikbasiert	semantikbasiert	semantikbasiert
Zweck	Unterstützung dynamischer Änderungen in Prozessinstanz-modellen	Nachvollzieh-barkeit von Änderungen	Ermittlung von Referenz-modellen	unklar	Ermittlung passender Prozesse	Identifkiation ähnlicher Prozessmodelle zur Unifikation
Vergleichs-kriterium für Prozesse	schwache Prozesslauf-ähnlichkeit	Korrespondenz-vollständigkeit von Fragmenten	Editierdistanz zwischen Prozess-modellen	Prozesslauf-ähnlichkeit	Editierdistanz zwischen Prozessläufen	Prozesslauf-ähnlichkeit
Vergleich-kriterium für Aktivitäten	Aktivitätstyp	Aktivitätsname	Aktivitätsname	Aktivitätsname	Aktivitätsname (Vokabular vorausgesetzt)	vermutlich Aktivitätsname
Eingabe						
Prozess-definitions-modell	ja	ja	ja	ja	ja	ja
Prozess-instanzmodell	ja	nein	nein	ja	nein	nein
Kardinalität Prüfling:Prüfer	1:n	1:1	1:n	1:1	1:1	1:1
Ergebnis						
qualitatives Ergebnis	nein	nein	nein	nein	nein	nein
quantitatives Ergebnis	ja (skalar)	nein	ja (skalar)	ja (vektorwertig)	ja (matrixwertig)	ja (vektorwertig)
detailliertes Ergebnis	ja (Konsistenz-bericht)	ja (change log)	nein	nein	nein	nein

Tabelle 5.1: Prozessmodellvergleichsansätze

5.8 Verwandte Arbeiten

Es existiert eine Reihe von Arbeiten, die den Vergleich von Prozessmodellen thematisieren. Ein bewertender Vergleich ist deshalb schwierig, weil die *Zielsetzung* der Vergleiche sich teils deutlich von dem Ansatz dieser Arbeit unterscheidet oder die Verwendung des Vergleichsergebnisses offengelassen wird. Zudem verfolgen andere Arbeiten gänzlich verschiedene algorithmische Ansätze.

In diesem Abschnitt werden verwandte Arbeiten beschrieben und mit dem Ansatz dieses Kapitels verglichen. Tabelle 5.1 fasst wichtige, im Weiteren aufgegriffene Vergleichskriterien zusammen. Um den Vergleich zu systematisieren, wird dabei zunächst eine Reihe von Eigenschaften beschrieben, die in den unterschiedlichen Ansätzen unterschiedlich ausgeprägt sind.

5.8.1 Kriterien für Prozessmodellvergleiche

Verwandte Arbeiten lassen sich untereinander und mit unserem Ansatz anhand einer Reihe von Kriterien vergleichen. Allen Prozessmodellvergleichen ist gemein, dass sie eine *Eingabe* (i.A. zwei Prozessmodelle) mittels eines *Verfahrens* auf eine bestimmte Art von *Ausgabe* (Vergleichsergebnis) abbilden, wonach der folgende Kriterienkatalog gegliedert ist.

Art der Eingabe (Prozessmodelle)

Prozessmodellierungssprache Die Sprache, in der die zu vergleichende Prozessmodellierungssprachen verfasst sind, beeinflusst je nach verwendetem Vergleichsverfahren die algorithmische Umsetzung teils erheblich. Für gängige Prozessmodellierungssprachen gibt es Arbeiten, deren Gegenstand der Vergleich von Modellen in der jeweiligen Sprache ist. Dazu gehören beispielsweise Ereignisgesteuerte Prozessketten [NR02], Petri-Netze [Rei86] und nichtstandardisierte, werkzeugspezifische Sprachen.

Berücksichtigte Aspekte Verwandt mit der Festlegung der Prozessmodellierungssprache ist, welche Aspekte in den Prozessmodellen festgehalten sind und beim Vergleich berücksichtigt werden. Die meisten Ansätze vergleichen Prozessdefinitionsmodelle ohne Berücksichtigung eventuell vorhandener Prozesshistorien.

Art der Ausgabe (Vergleichsergebnis)

Die untersuchten verwandten Ansätze unterscheiden sich stark in der Detailliertheit ihrer Ergebnisse.

Qualitative Ansätze Die denkbar abstrakteste Form bildet eine *qualitative, zweiwertige* Ausgabe, aus der nur abgelesen werden kann, ob zwei eingegebene Prozessmodelle gemäß einem bestimmten Begriff ähnlich sind oder nicht. Weder unser noch einer der untersuchten verwandten Ansätze hat diese Art von Ergebnis zum Ziel, was dadurch begründet ist, dass ein rein zweiwertiges Ergebnis bei Prozessmodellvergleichen keinerlei Nutzen für Prozessmodellierer darstellt.

Quantitative Ansätze Die überwiegende Anzahl verglichener Ansätze quantifiziert die Ähnlichkeit bzw. Unähnlichkeit von Prozessmodellen [LRW08b, AMW06, MAW08, WR06, Dij08]. Das Ergebnis trifft also Aussagen über das

Ausmaß der vorgefundenen Ähnlichkeit von Prozessmodellen. Hierbei kann unterschieden werden zwischen Ansätzen, die die Ähnlichkeit als Skalarwert ausdrücken, wie beispielsweise unser Ansatz und der von Li et al. [LRW08a], oder unterschiedliche Ähnlichkeitsmetriken unterscheiden und ein vektorwertiges Ergebnis liefern, wie der Ansatz von Dijkman und Aalst et al. [Dij08, AMW06].

Unterschiedsdetaillierende Ansätze Nur wenige Ansätze wie beispielsweise unserer oder [KGFE08] sehen vor, die Unterschiede zwischen Prozessmodellen in so detaillierter Form zu berechnen, dass aus dem Ergebnis einzelne, ggf. schwer erkennbare Unterschiede hervorgehen. Stattdessen werden bei vielen Ansätzen komplexe Zwischenergebnisse berechnet, die hierfür aufbereitet werden könnten, jedoch schließlich zu einem Skalar bzw. Vektor verdichtet werden.

Art des Vergleichsverfahrens

Vergleichsbasis Existierende Verfahren zu Berechnung von Prozessmodellähnlichkeiten können grundsätzlich unterteilt werden in solche, die syntaktische Unterschiede zwischen Prozessmodellen berechnen und solche, die die Semantik, also die semantische Domäne von Prozessmodellen als Ausgangspunkt für Vergleiche verwenden.

Syntaxbasierte Verfahren Bei syntaxbasierten Ansätzen werden Prozessmodelle als Ausprägungen einer bestimmten Graphklasse betrachtet. Es steht also die reine Struktur im Vordergrund. Der *Vorteil* dieser Ansätze gegenüber semantikbasierten ist, dass detaillierte Ergebnisse von Berechnungen auf der syntaktischen Struktur eines Prozessmodells leicht in eine Form zu bringen sind, die sich wiederum in die syntaktische Prozessmodellstruktur abbilden lässt. Dieser Vorteil wird beispielsweise in [KGFE08] ausgenutzt. Ein weiterer Vorteil ist, dass syntaxbasierte Verfahren auf kleineren Datenstrukturen arbeiten und somit generell eine geringere Speicherplatz- und Laufzeitkomplexität aufweisen. Ein *Nachteil* dieser Verfahren ist, dass syntaktisch sehr unterschiedliche Prozessmodelle auch bei eng gefassten Ähnlichkeitsbegriffen eine identische oder zumindest sehr ähnliche Semantik modellieren und umgekehrt geringe Unterschiede in der Prozessmodellsyntax sich in großen semantischen Unterschieden niederschlagen können [AMW06]. Letztlich ist der semantische Unterschied für den Prozessmodellierer von Interesse und nicht, wie ggf. ein und derselbe Prozess in syntaktisch unterschiedlicher Weise modelliert werden kann. Ausnahmen von dieser Annahme bilden solche Ansätze, die insbesondere die Auffindung von Refactoring-Maßnahmen in Prozessmodellrevisionen bezwecken, wie beispielsweise [KGFE08].

Semantikbasierte Ansätze Gegenstand von semantikbasierten Verfahren bilden die semantischen Domänen von Prozessmodellen. Semantische Domänen können dabei, wie in dieser Arbeit, Transitionssysteme sein oder formal sehr ähnliche endliche Automaten, aber auch Mengen von Zeichenketten, die mögliche Prozessläufe repräsentieren. Solche Ansätze erlauben es, von syntaktischen Details der jeweiligen Prozessmodellierungssprache zu abstrahieren, beispielsweise in WS-BPEL davon, ob die Ausführungspräzedenz zwischen zwei Aktivitäten über die Anordnung innerhalb eines `Sequence`-Elementes definiert wurde oder über ein `Link`-Element innerhalb eines `Flow`-Elementes. Bei Übertragung auf eine neue Prozessmodellierungssprache muss zudem im Wesentlichen nur die Abbildung in die semantische Domäne angepasst werden, die Algorithmen, die auf der semantischen Domäne basieren, hingegen nicht. Offenkundig wird bei diesen Verfahren auch das Problem umgangen, dass der gleiche Prozess syntaktisch unterschiedlich modelliert werden kann. Rein syntaktische Unterschiede in Prozessmodellen werden durch Abbildung in eine semantische Domäne nivelliert.

Vergleichskriterien

Vergleichskriterium für Aktivitäten In jedem Ansatz wird zumindest als Vorbereitung des eigentlichen Vergleichs die Syntax der Prozessmodelle analysiert. Dabei bilden bekanntermaßen atomare Aktivitäten die syntaktisch niedrigste Ebene, unterhalb derer keine Details in den Prozessmodellen zu finden sind. Dadurch stellt sich die generelle Frage, wann atomare Aktivitäten als gleich oder ungleich anzusehen sind. Viele Ansätze ignorieren diese Fragestellung und setzen voraus, dass gleichbedeutende Aktivitäten identisch benannt sind. Praxisnahe Prozessmodelle beinhalten jedoch i.A. Synonyme, die als solche erkannt bzw. (beispielsweise über einen Typen) ausgezeichnet werden müssen. Des Weiteren kann nicht davon ausgegangen werden, dass eine Aktivität genau einmal in einem verglichenen Prozessmodell vorkommt, was gerade bei syntaxbasierten Vergleichsverfahren beachtet werden muss.

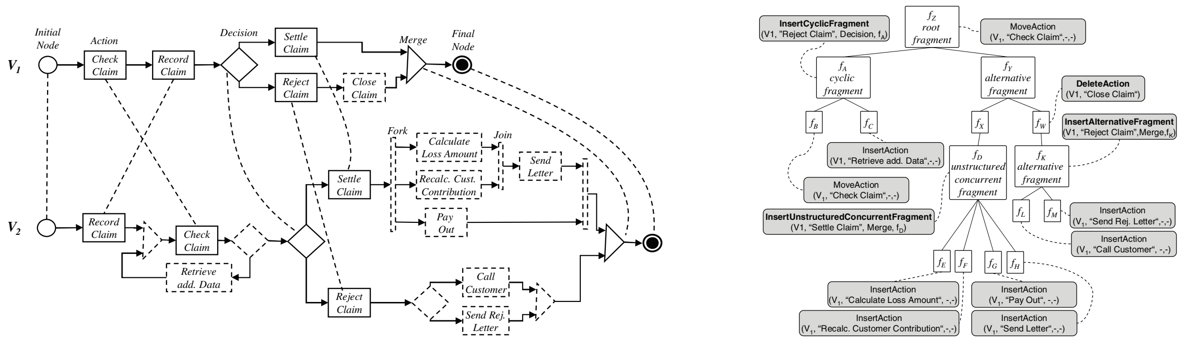
Vergleichskriterien für Prozesse Neben der Entscheidung, ob ein Vergleichsalgorithmus auf Basis der Syntax oder semantischen Domäne eines Prozessmodells entwickelt werden soll, muss auch festgelegt werden, welche Kriterien erfüllt sein müssen, damit Prozessmodelle als gleich oder zumindest ähnlich angesehen werden. Bei syntaxbasierten Ansätzen bieten sich hierfür im Wesentlichen nur Graphisomorphietests, wie ursprünglich von [CG70] beschrieben, zur Berechnung qualitativer Ergebnisse und Grapheditierdistanzen [Bun97, Definition 8] oder Untergraphisomorphietests [MB99, MB98] für quantitative Ergebnisse auf der syntaktischen, graphischen Prozessmodellstruktur an. Bei semantikbasierten Ansätzen, deren semantische Domäne wie in unserem Fall ein Transitionssystem ist, gibt es eine ganze Reihe von einander implizierenden Kriterien, die sich in zwei Gruppen unterteilen lassen. Bisimilaritätskriterien stellen Bedingungen an die Verzweigungsstruktur der Transitionssysteme, Prozesslaufkriterien an deren Pfade:

(Starke) Bisimilarität ist nach Milner [Mil80, Abschnitt 3.3] Folgendes: Seien $\mathfrak{T}_p$ und $\mathfrak{T}_q$ zu Prozessinstanzmodellen gehörige Transitionssysteme (vgl. Abschnitt 2.5), wobei die Transitionssysteme nicht zwangsweise Graphtransitionssysteme sein müssen. Starke Similarität erfordert verknappt formuliert, dass eine Äquivalenzrelation $\cong$ zwischen Zuständen aus $\mathfrak{T}_p$ und $\mathfrak{T}_q$ existiert, so dass für jeden Zustandsübergang $s_p \xrightarrow{\text{a}} s'_p$ in p und $s_p \cong s_q$ ein Zustandsübergang $s_q \xrightarrow{\text{a}} s'_q$ existiert mit $s'_p \cong s'_q$. Gewissermaßen müssen also alle durch ein a bewirkte Zustandswechsel in p in q nachvollzogen werden können. Die Startzustände sind per Definition äquivalent. Können umgekehrt auch Zustandswechsel in q innerhalb von p nachvollzogen werden, sind p und q stark bisimilar.

Schwache Bisimilarität Bei der starken Bisimilarität wird gefordert, dass ein a-Übergang in $\mathfrak{T}_p$ unmittelbar in $\mathfrak{T}_q$ nachvollzogen werden muss. Schwache Similarität nach Glabbeek et al. [Gla90, Definition 2.3] fordert, dass für jeden Zustandsübergang $s_p \xrightarrow{\text{a}} s'_p$ in p und $s_p \cong s_q$ eine Kette von Zustandsübergängen $s_q \xrightarrow{\tau_1} s'_q \xrightarrow{\tau_2} \dots \xrightarrow{\tau_k} s''_q \xrightarrow{a} s'''_q \xrightarrow{\tau_{k+1}} \dots \xrightarrow{\tau_n} s''''_q$ existiert, mit $s'_p \cong s''''_q$. Die τ_i sind spezielle Transitionsbeschriftungen, so genannte "silent moves", die als irrelevant betrachtet werden. Es reicht also in $\mathfrak{T}_q$, über einen Pfad zum zu s'_p äquivalenten Zustand zu gelangen, auf dem genau ein a-Übergang existiert und sonst nur "silent moves".

Vollständige Prozesslaufgleichheit Es liegt eine vollständige Prozesslaufgleichheit zwischen zwei Prozessinstanzmodellen p und q vor, wenn in beiden die gleichen Prozessläufe möglich sind, d.h. nach Definition 5.1 $c(p) = c(q)$ gilt. Dieses Kriterium geht zurück auf die so genannte "trace equivalence" von Prozessen nach [Pnu85].

Teilweise Prozesslaufgleichheit Eine teilweise Prozesslaufgleichheit zwischen zwei Prozessinstanzmodellen p und q gilt, wenn $c(p) \cap c(q) \neq \emptyset$, also mindestens ein gemeinsamer Prozesslauf existiert. Unter den genannten ist sie das schwächste Kriterium.



(a) Korrespondenzen zwischen V_1 und V_2 (b) Hierarchisches “change log”

Abbildung 5.14: Berechnung eines Change-Logs zwischen Prozessdefinitionsmodellen V_1 und V_2 [KGFE08]

5.8.2 Syntaxbasierte Ansätze

In diesem Unterabschnitt werden einige syntaxbasierte Verfahren erörtert und mit unserem semantikbasierten Ansatz verglichen.

Inferenz von Änderungsoperationen (Küster et. al)

In [KGFE08] beschreiben Küster et. al, wie Differenzen zwischen Prozessmodellversionen algorithmisch auf Basis der Prozessmodellsyntax ermittelt werden können.

Zweck Zweck des Ansatzes ist, aus den berechneten Unterschieden a-posteriori eine Editierfolge zu berechnen, mit denen eine Prozessdefinitionsmodellversion V_1 in eine Prozessdefinitionsmodellversion V_2 überführt werden kann, wodurch sich der Ansatz besonders für solche Fälle eignet, in denen V_2 tatsächlich eine (verbesserte) Revision von V_1 ist. Die Editierfolge wird dem Prozessmodellierer schlussendlich in Form eines hierarchischen “change logs” präsentiert, wobei die Hierarchie die Enthaltenseinsbeziehung zwischen Aktivitäten beider Prozessmodelle ist.

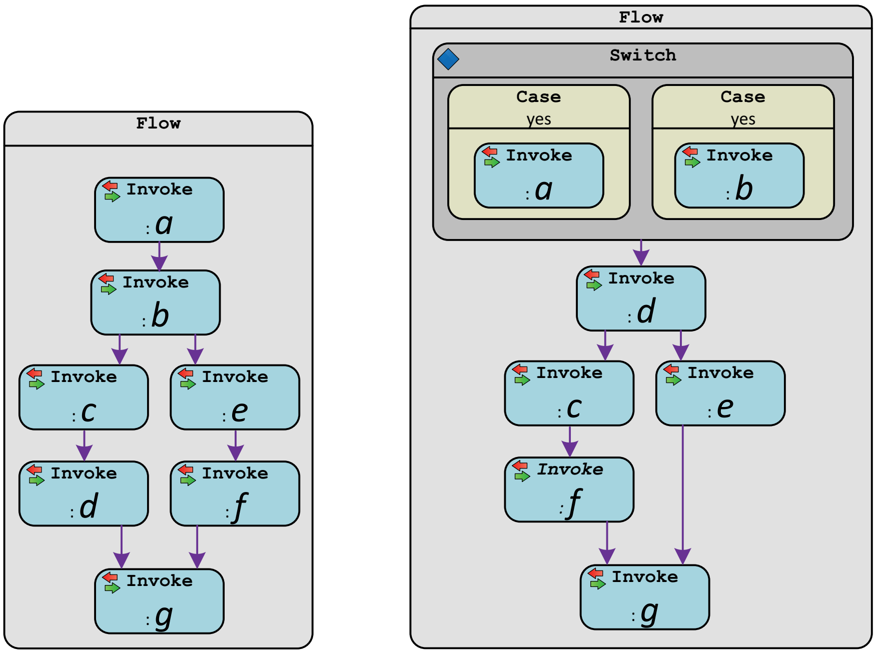
Ansatz Die verglichenen Prozessdefinitionsmodelle sind in einer proprietären Prozessmodellierungssprache modelliert, die im IBM WebSphere Business Modeler [WAM+07, Kapitel 5] als Vorstufe für WS-BPEL-Prozessdefinitionsmodelle editiert werden können. Da aus Modellen dieser Sprache die Enthaltenseinshierarchie von Aktivitäten nicht direkt hervorgeht, muss diese im ersten Schritt nachträglich berechnet werden, wobei komplexe Aktivitäten als Single-Entry-Single-Exit-Fragments (SESE-Fragments) bezeichnet werden. Im zweiten Schritt werden SESE-Fragmente mit syntaktisch identischem Inhalt als Korrespondenzen identifiziert. Anschließend werden auf Basis der Korrespondenzen im dritten Schritt nicht-korrespondierende Teile als eingefügt bzw. gelöscht identifiziert, sowie aus der Kontrollflussdefinition zzgl. der Korrespondenzen Verschiebungen vorhandener Aktivitäten erkannt. Die erkannten Einfüge-, Lösch- und Verschiebeoperationen werden – zusammen mit den jeweils betroffenen Aktivitäten – als Blätter (grau dargestellt in Abbildung 5.14) in einen Baum eingefügt, der die Vereinigung der Enthaltenseinshierarchien von V_1 und V_2 ist. Dieses sogenannte "change log" wird dem Prozessmodellierer letztlich angezeigt.

Gemeinsamkeiten Der Ansatz liefert detaillierte statt nur rein qualitative oder quantitative Vergleichsergebnisse und ähnelt insoweit dem zuvor in diesem Kapitel beschriebenen Ansatz.

Unterschiede Im Unterschied zu unserem Ansatz wird der Vergleich nur für zwei ausgewählte Paare von Prozessdefinitionsmodellen durchgeführt. Es findet also keine mit ähnlichen Mitteln durchgeführte Auswahl (t-)ähnlicher Prozessdefinitionsmodelle statt. Der Prozessmodellierer muss folglich wissen oder auf anderem Wege berechnen lassen, welche Prozessdefinitionsmodelle sinnvollerweise miteinander verglichen werden können. Der Ansatz von Küster et al. trifft außerdem keine Aussage darüber, wie Umbenennungen von Aktivitäten in einem Prozessdefinitionsmodell behandelt werden, beispielsweise von "Calculate Losses" in "Calculate Loss Amount". Es ist zu vermuten, dass die Korrespondenzfindung die Identität von Aktivitätsnamen voraussetzt. In unserem Ansatz ist der Aktivitätsname irrelevant, d.h. ohne Auswirkung frei wähl- und änderbar und nur der jeweilige von Aktivitäten referenzierte Aktivitätstyp wichtig, der nur an einer Stelle benannt wird, allerdings mit den in Unterabschnitt 4.6.2 erläuterten Nachteilen. Der Ansatz von Küster et al. basiert im Unterschied zu unserem durchgängig auf der Prozessmodellsyntax mit den bereits zuvor genannten Vor- und Nachteilen

Editierdistanzen in Prozessdefinitionsmodellen (Li et al.)

In [LRW08b] beschreiben Li et al. einen ebenfalls syntaxbasierten jedoch quantitativen Ansatz zum Vergleich von Prozessdefinitionsmodellen, die in der Sprache der "Kontrollflussgraphen" [Rei00] verfasst sind. Prinzipiell können Kontrollflussgraphen auch Prozessinstanzmodelle darstellen, was in [LRW08b] jedoch unberücksichtigt bleibt.



(a) rl_1 (b) rl_2

Abbildung 5.15: Syntaktische ähnliche Prozessdefinitionsmodelle (nach [LRW08b])

Zweck Der entwickelte Vergleichsalgorithmus wird im Bereich der Prozessmodellinferenz ("process mining") [ADH+03] eingesetzt. Hierbei soll für eine Menge von ähnlichen Prozessdefinitionsmodellen ein Referenzdefinitionsmodell identifiziert werden, dessen Ähnlichkeit zu den übrigen maximal ist [LRW08a].

Ansatz Für ein Prozessdefinitionsmodell p wird eine Ausführungsreihenfolgematrix A^p berechnet, in der Reihen bzw. Spalten den Aktivitäten von p bzw. q zugeordnet sind. Ein Eintrag A^p_{ij} ist 1 bzw. 0, falls Aktivität a_i immer vor bzw. nach a_j durchgeführt wird. Falls Prozessläufe für den ersten und den zweiten Fall existieren, ist $A^p_{ij} = *$; falls kein Prozesslauf existiert, in dem sowohl a_i als auch a_j durchgeführt wird, ist $A^p_{ij} = -$. Für zwei Prozessdefinitionsmodelle p und q wird anschließend eine Menge minimaler Größe von Einfüge-, Lösch- und Verschiebeoperationen berechnet, um $A^p_{ij} = A^q_{ij}$ für alle i und j zu erreichen, die also alle Konflikte zwischen p und q bzgl. Vorhandensein und Ausführungsreihenfolge von Aktivitäten beseitigt. Der Vergleich der Prozessdefinitionsmodelle aus Abbildung 5.15, die der Einfachheit halber in die in dieser Arbeit verwendeten Sprache SimBPEL übersetzt wurden, liefert hierfür beispielsweise fünf minimal benötigte Änderungsoperationen.

Unterschiede Auch dieser Ansatz basiert auf der Annahme, dass Aktivitäten eindeutig und unveränderlich sind und ignoriert somit, dass Aktivitätsnamen in der Praxis durchaus häufig geändert werden bzw. Aktivitätsnamen homonym oder synonym gebraucht werden können. Die Beispiele sind daher auch dergestalt, dass jede Aktivität genau einmal im Prozessdefinitionsmodell modelliert ist, was in praxisnahen Modellen normalerweise nicht der Fall ist. Dadurch, dass der Ansatz von Li et al. auf der Modellsyntax basiert, ergeben sich auch unterschiedliche Ähnlichkeitsbegriffe. Dies äußert sich darin, dass die bilaterale Prozessunähnlichkeit gemäß unseres Ansatzes $d(rl_1, rl_2) = 3$ ist, die Modelle sich unter unserem Begriff also recht ähnlich sind, da sie sich nur darin unterscheiden, dass erstens in rl_1 sowohl die a-Aktivität, als auch die b-Aktivität ausgeführt werden muss, in rl_2 jedoch nur eine von beiden und zweitens, dass in rl_1 die c-Aktivität vor der d-Aktivität ausgeführt und in rl_2 andersherum die d-Aktivität vor der c-Aktivität ausgeführt werden muss.

5.8.3 Semantikbasierte Verfahren

Semantikbasierte Verfahren überführen die syntaktische Struktur eines Prozessmodells zunächst in eine semantische Domäne (Transitionssysteme, Automaten etc.) und verwenden diese für weitere Analysen.

Modellvergleiche mithilfe von Prozesshistorien (Aalst et al.)

Aalst et al. beschreiben in [AMW06, MAW08] einen semantikbasierten Ansatz zum quantitativen Vergleich von Prozessdefinitionsmodellen. Hauptmerkmal des Ansatzes ist, dass zwei Prozessdefinitionsmodelle anhand ihrer Prozessläufe verglichen werden. Dabei können nicht nur theoretisch erzeugbare Prozessläufe gleichberechtigt einbezogen werden, sondern zusätzlich die Häufigkeitsverteilung einzelner, aufgezeichneter, real durchgeführter Prozessläufe miteinbezogen werden. Der Ansatz basiert nicht auf einer bestimmten Prozessmodellierungssprache sondern auf dem Formalismus der Petri-Netze (vgl. Abschnitt 2.4).

Zweck Einen bestimmten Zweck ist der Vergleichsansatz von Aalst et al. nicht gewidmet. Vielmehr wird behauptet, dass der quantitative Vergleich von Prozessmodellen von generellem Interesse ist.

Ansatz Ausgehend von einer Menge von aufgezeichneten Prozessläufen $L \subseteq c(p)$ eines Prozessdefinitionsmodells p und einem Prozessdefinitionsmodell q werden mehrere Kennzahlen berechnet

Fitness $fitness(L, q) \in [0, 1]$ gibt an, in welchem Ausmaß die Prozessläufe in L durch q auch erzeugt werden könnten. Dabei bedeutet der Extremwert 0, dass in keinem einzigen Prozesslauf $l \in L$ auch nur eine Aktivität so positioniert ist, dass sie an ihrer Position durch q durchgeführt werden könnte. Ein Wert von 1 bedeutet, dass $L \subseteq c(q)$ ist, also jeder Prozesslauf in L vollständig auch in q durchgeführt werden kann.

Precision $precision(p, q, L) \in [0, 1]$ gibt an, wie viele Aktivitäten nach Abarbeitung von $l \in L$ sowohl in p als auch in q durchführbar sind im Verhältnis zu den Aktivitäten, die nur in p durchführbar sind. Weitere, hier nicht beschriebene Divisoren bewirken eine Normalisierung auf einen reellen Wert zwischen 0 und 1. Der Wert erreicht 1, wenn q zu jedem Zeitpunkt der Abarbeitung eines der gegebenen Prozessläufe größere Wahlfreiheit bzgl. möglicher Prozessläufe als p lässt und sinkt für jeden Fall, in dem dies nicht zutrifft.

Recall $recall(p, q, L) = precision(q, p, L)$ ist die umgekehrte Betrachtung mit vertauschtem p und q.

In [MAW08] sind die Mengen so beschrieben, dass auch eine algorithmische Berechnung daraus direkt hervorgeht.

Gemeinsamkeiten Gemein mit unserem Ansatz ist, dass Unterschiede quantifiziert werden könnten und das Verhalten anstatt der reinen Prozessmodellsyntax die Basis bildet. Ebenso beziehen Aalst et al. Prozesshistorien in die Berechnungen mit ein, wenn auch als separate Objekte und nicht als Bestandteil von Prozessinstanzmodellen wie in unserem Fall.

Unterschiede Im Unterschied zu unserem Ansatz sind in dem von Aalst et al. Prozessläufe unbedingt erforderlich, um zwei Prozessdefinitionsmodelle überhaupt miteinander vergleichen zu können. Eine leere Prozesslaufmenge führt zu undefinierten Ergebnissen, d.h. gewisse Faktoren nehmen den Wert $\frac{0}{0}$ an. In unserem Ansatz sind Prozessläufe als Prozesshistorien optional und führen bei Vorhandensein nur zu geringerer Wahlfreiheit bei der pseudosimultanen Prozessdurchführung und somit im Normalfall zu einer Erhöhung der bilateralen Prozessunähnlichkeit. Demgegenüber berücksichtigen Aalst et al. die Tatsache, dass Prozesshistorien mehrfach auftreten können. Die Häufigkeit des Vorkommens einer Prozesshistorie wird direkt verrechnet, in unserem Ansatz nicht, was spätestens bei der Detailanalyse durchaus zu Redundanzen führen kann. Ein weiterer Unterschied zu unserem ist, dass der Ansatz von Aalst et al. neben der Ähnlichkeitsquantifizierung keine weiteren Details bzgl. des Unterschieds zweier Prozessinstanzmodelle liefert.

Editierdistanzen zwischen Prozessläufen (Wombacher et al.)

Vergleichbar mit der zuvor beschriebenen Arbeit von Aalst et al., die sich auf einem Formalismus (Petri-Netze) statt einer bestimmten Prozessmodellierungssprache bezieht, fußen die Ergebnisse von Wombacher et al. [WR06] auf dem Formalismus endlicher Automaten [HMU02]. Strukturvergleiche zwischen Automaten, die Syntaxvergleichen zwischen Prozessmodellierungssprachen entsprechen, werden dabei auch als unzureichend erkannt. Die Basis des Ansatzes von Wombacher et al. sind daher die Sprachen, die zwei zu vergleichende Automaten erzeugen.

Zweck Die Anwendung ihrer Überlegungen sehen die Autoren im Umfeld der Dienstsuche ("service discovery") im Bereich der WebServices. Bei zwei zu vergleichenden Automaten p und q nimmt beispielsweise p die Rolle einer Anfrage ein, die ein gesuchtes Verhalten formuliert, q hingegen ein Element aus dem Suchraum, welches Element der Ergebnismenge ist, sollten p und q hinreichend ähnlich sein. Derartige Anfragen sind relevant, wenn beantwortet werden soll, ob das mitunter komplexe Außenverhalten eines WebServices so zu einem anderen passt, dass diese miteinander interagieren können.

Ansatz Der Ansatz von Wombacher et al. basiert auf Distanzen zwischen Zeichenketten $w \in \mathcal{L}(p)$ und $w' \in \mathcal{L}(q)$, wobei $\mathcal{L}(p)$ und $\mathcal{L}(q)$ die Sprachen sind, die die zu vergleichenden Automaten p und q akzeptieren. Ergebnis eines Vergleichs zwischen p und q ist eine Matrix mit den Dimensionen $|\mathcal{L}(p)| \times |\mathcal{L}(q)|$, wobei jeder Eintrag die Levenshtein-Distanz zwischen zwei Zeichenketten w und w' ist. Bei zyklischen Automaten, die unendliche Sprachen mit beliebig langen Wörtern akzeptieren, wird unter Inkaufnahme von geringerer Präzision die Levenshtein-Distanz von n-grammen [BY92] berechnet, wobei ein n-gramm, vergleichbar mit einem regulären Ausdruck, wiederum eine (Unter-)Sprache repräsentiert.

Gemeinsamkeiten Die Gemeinsamkeiten mit unserem Ansatz sind in erster Linie in der formalen Basis der Levenshtein-Distanzen zwischen akzeptierten Zeichenketten bzw. Prozessläufen zu sehen. Beide Ansätze berücksichtigen unendliche Mengen von Prozessläufen, die mit zyklische Automaten bzw. `While`-Aktivitäten einhergehen. Wombacher et al. erkennen zudem, dass ein Vokabular von eindeutigen, unveränderlichen Aktivitätsnamen vorausgesetzt werden muss, wobei ihr Ansatz keinerlei Hilfsmittel bereitstellt, dieses zu unterstützen.

Unterschiede Der Unterschied zwischen unserem Ansatz und dem von Wombacher et al. liegt wiederum darin, dass die Ergebnisse über eine Quantifizierung nicht hinausgehen. Auffällig ist insbesondere, dass der Vergleich zweier Automaten matrixwertig ist; eine Verdichtung der Matrix auf einen skalaren Wert, der die allgemeine Ähnlichkeit zwischen beiden Automaten statt der paarweisen Ähnlichkeit der Zeichenketten darstellt, wird nicht beschrieben. Ein weiterer Unterschied ist, dass Wombacher Levenshtein-Distanzen direkt auf Zeichenketten bzw. n-grammen berechnet, wohingegen unser Ansatz dies nur indirekt mittels Graphersetzungen und anschließender Pfadanalyse mit Fokus auf geringste Distanzen bzw. günstigste Pfade bewerkstelligt.

Vergleiche reduzierter Prozessläufe (Dijkman)

Dijkman erläutert in [Dij08] (sowie [Dij07]) einen Ansatz zur Quantifizierung von Unterschieden zwischen Prozessdefinitionsmodellen, die in EPC ("event process chains" [NR02]) modelliert sind. Das Ergebnis ist hierbei ein Vektor, dessen Komponentenwerte Unterschiede gemäß bestimmter Kriterien quantifizieren.

Zweck Dijkman bezweckt mit Vergleich von Prozessdefinitionsmodellen die Unifikation ähnlicher Prozesse. Solche Unifikationen sind beispielsweise bei Reorganisationsmaßnahmen innerhalb von Firmen, aber insbesondere auch bei Firmenzusammenschlüssen notwendig.

Ansatz Zur Berechnung der Unterschiede werden EPC-Prozessdefinitionsmodelle in endliche Automaten übersetzt. Semantisch gleiche Aktivitäten müssen danach manuell identifiziert werden, und werden mit einem eindeutigen, gemeinsamen Bezeichner versehen. Anschließend werden die Automaten mit Bezug auf eine bestimmte, gemeinsame Aktivität x jeweils so minimiert, dass zu x alternative und nachgelagerte Aktivitäten in den minimierten Automaten nicht enthalten sind. Die erzeugten Sprachen $\mathcal{L}_1, \mathcal{L}_2$ werden dann miteinander im Hinblick auf bestimmte Kriterien verglichen, beispielsweise, ob ein Wort in $\mathcal{L}_1$ existiert, das mit x endet, welches in $\mathcal{L}_2$ nicht vorkommt und umgekehrt. Die Feststellung in diesem Fall lautet, dass x in den Prozessdefinitionsmodellen unter verschiedenen Bedingungen ("different conditions") ausgeführt wird.

Gemeinsamkeiten Dijkman erkennt, dass Aktivitäten in unterschiedlichen Prozessmodellen nicht allein über die eventuelle Namensgleichheit korreliert werden können. Wie auch in unserem Ansatz, ist die Korrelation letztlich manuell durchzuführen, um brauchbare Ergebnisse zu erhalten. Der Bezug auf bestimmte Aktivitäten pro Vergleich ähnelt der Verwertung des Detailanalyse im Konsistenzbericht (vgl. Abschnitt 5.6). Dabei findet in unserem Ansatz die Ausrichtung der Zwischenergebnisse auf eine bestimmte Aktivität erst in späteren Phasen statt und nicht etwa, vergleichbar mit dem Ansatz von Dijkman, in der Vorbereitungsphase innerhalb des Transitionssystems.

Unterschiede Zwar sind die Vergleichsergebnisse nach dem Ansatz von Dijkman nach bestimmten Kriterien aufgeschlüsselt, geben aber keine Rückschlüsse auf konkrete Unterschiede verglichener Prozessdefinitionsmodelle. Wiederum werden nur Prozessdefinitions- aber keine -instanzmodelle berücksichtigt.

Kapitel 6

Schlussbemerkungen

Dieses Kapitel fasst die *Beiträge* dieser Arbeit in Abschnitt 6.1 kurz zusammen. Abschnitt 6.2 zeigt weitere *Fragestellungen* auf, die durch diese Arbeit aufgeworfen aber nicht beantwortet wurden.

6.1 Zusammenfassung

Ausgangspunkt dieser Arbeit war die Problemstellung des kooperierenden Industriepartners Generali Deutschland Informatik Services GmbH – der IT-Dienstleister für Versicherungsunternehmen der Generali Gruppe. Die Generali Deutschland Informatik Services GmbH setzt zur Unterstützung von Versicherungsprozessen das Prozessmanagementsystem *IBM WebSphere Process Server* ein. Als Prozessmanagementsystem dient dieses der Steuerung von Versicherungsprozessen, beispielsweise der Abwicklung von Leitungswasserschäden. Es versorgt am Prozess beteiligte Personen und Systeme zum richtigen Zeitpunkt mit den richtigen Daten und führt dabei Buch über die Prozessdurchführung. Der IBM WebSphere Process Server gehört zu der verbreiteten Klasse von Prozessmanagementsystemen, die die *Modellierung* eines Prozesses *nur vor* dessen *Ausführung* zulassen. Diese Unzulänglichkeit fördert erstens die Bildung von so genannten *Pseudo-Maximalmodellen*, die hochkomplex und daher schlecht wartbar sind und in denen beabsichtigt wird, alle denkbar sinnvollen Abläufe festzuhalten. Da dies nicht gelingen kann, kommt es häufig vor, dass Prozesse mit unvorhergesehenen Abläufen entkoppelt vom Process Server durchgeführt werden. Damit entfällt nicht nur die Unterstützung für solche Prozesse, auch die spätere Nachvollziehbarkeit von Prozessabläufen ist in solchen Fällen schwerer möglich.

In dieser Arbeit wurde ein neuartiges, dynamisches Prozessmanagementsystem beschrieben, über das dynamische Geschäftsprozesse unterstützt werden können. Die Dynamik in Geschäftsprozessen kann damit so behandelt werden, dass bei gegebenen Anlass ein Prozessbeteiligter *Ablaufänderungen* in einem laufenden Prozess vornimmt, die anschließend wirksam für die weitere Prozessdurchführung sind. Unterstütze Arten von Änderungen sind in dieser Arbeit das dynamische Einfügen und Löschen von Aktivitäten sowie Rücksprünge zu bereits (fehlerhaft) ausgeführten Prozessteilen.

Im Vergleich zu verwandten Arbeiten ist das neue, dynamische Prozessmanagementsystem insoweit einzigartig, als dass es nicht von Grund auf entwickelt wurde, sondern den IBM WebSphere Process Server lediglich *erweitert*. Damit bleiben dessen zahlreichen Funktionen, Integrationsmöglichkeiten und Qualitätsmerkmale erhalten. Die Erweiterung ist *nicht-invasiv*, d.h. die Implementierung des Process Server sollte und konnte nicht geändert werden. Sie besteht aus weiteren Modellierungssprachen, die die vorhandene Prozessmodellierungssprache WS-BPEL ergänzen, einer zusätzlichen *Software-Schicht* oberhalb des WebSphere Process Servers und einem Modellierungswerkzeug, das Prozessbeteiligte bei Änderungen an Prozessmodellen durch *Modellprüfungen* unterstützt.

6.1.1 Prozessmodelle

Lediglich *Prozessdefinitionsmodelle*, in denen ein bestimmter Prozesstyp in der verbreiteten Sprache WS-BPEL modelliert wird, können vom IBM WebSphere Process Server verarbeitet werden. In dieser Arbeit wurden Prozessdefinitionsmodelle ergänzt um *Prozessinstanzmodelle*, die zusätzlich zur Ablaufstruktur den aktuellen Prozesszustand und bisherigen Ablauf modellieren. Prozessinstanzmodelle stellen Prozesse somit ganzheitlich dar und taugen für die Umsetzungen von *dynamischen Änderungen* im Gegensatz zu der sonst üblichen, ausschnitthaften und tabellarischen Darstellung laufender Prozesse. *Prozesswissensmodelle* abstrahieren von genauen Kontrollflussdefinitionen zwischen Aktivitäten, die sich in Prozessdefinitions- und -instanzmodellen wiederfinden. In ihnen wird in abstrakter Form allgemeines *Wissen über Prozessabläufe* festgehalten, beispielsweise dass der Beauftragung eines externen Gutachters in einer Schadensabwicklung auch (mittelbar) eine Aktivität folgen muss, die das Einpflegen des resultierenden Gutachtens bezweckt. Derartige Reihenfolgebedingungen, sowie Bedingungen, die den gegenseitigen Ausschluss bestimmter Aktivitäten oder das n-fache Vorhandensein einer Aktivität in einem Prozess fordern, können in Prozesswissensmodellen ausgedrückt werden. Die zusätzlichen Sprachmittel der *Spezialisierung* und *Multiplizitäten* an Bedingungen erhöhen die Ausdrucksmächtigkeit von Prozesswissensmodellen und vermindert ebenso Redundanzen. Sie sind in der Form in keiner vergleichbaren Sprache zu finden.

Die *Syntax* der verschiedenen Prozessmodellarten wurde durch ein *gemeinsames Metamodell* formal festgehalten. Durch die Verwendung modellgetriebener Entwicklungswerkzeuge konnten aus diesem Metamodell Diagrammeditoren für die einzelnen Prozessmodellarten *generiert* werden, die zu einem so genannten *Prozessmodelleditor* zusammengefasst sind.

Die *Semantik* der verschiedenen Prozessmodellarten wurde ebenfalls *formal definiert*. Die Semantik von *Prozessinstanzmodellen* und *Prozessdefinitionsmodellen* wurde durch eine *Graphgrammatik* unter Verwendung des Werkzeugs GROOVE formal definiert. Die Graphgrammatik besteht dabei aus einer festen Graphersetzungsregelmenge $\mathcal{R}_{bpel}$ zzgl. eines Startgraphs $\mathrm{pg}(p)$, der aus einem Prozessinstanzmodell gewonnen wird. Aus einer Graphgrammatik lässt sich ein *Transitionssystem* erzeugen, welches sämtliche Ausführungszustände eines Prozessinstanzmodells in Graphform und mögliche Übergänge beinhaltet. Derartige Transitionssysteme bilden u.A. die Grundlage für die Semantikdefinition von *Prozesswissensmodellen*. Die Semantik von Prozesswissensmodellen ist derart geklärt, dass einzelne Aussagen in diesen Modellen (Bedingungsbeziehungen) auf komplexe mathematische Ausdrücke abgebildet werden. Diese Ausdrücke treffen *Aussagen über* in Prozessinstanzmodellen enthaltene *Prozesshistorien* und, mittels Formeln der temporalen Logik, über besagte *Transitionssysteme*, die noch mögliche zukünftige Abläufe eines Prozessinstanzmodells beinhalten. Die Berücksichtigung der Prozesshistorie in der Semantikdefinition von Prozesswissensmodellen ist in verwandten Arbeiten nicht zu finden, ist aber für Aussagen über laufende Prozesse notwendig.

6.1.2 Erweiterung des WebSphere Process Servers

Der IBM WebSphere Process Server lässt dynamische Änderungen an laufenden Prozessen nicht zu. Die von Prozessdefinitionsmodellen zum Prozessstart abgeleiteten Prozessinstanzen können nur den Abläufen folgen, die im Prozessdefinitionsmodell zur Instanziierungszeit vorgesehen waren. Ausgehend von dieser technischen Einschränkung des WebSphere Process Servers, die auch in vergleichbaren Systemen zu finden ist, wurde der Process Server um eine so genannte *Dynamik-Schicht* erweitert, die dynamische Änderungen simuliert. Diese Schicht besteht aus *zwei Komponenten*: Zur Modellierungszeit werden Prozessdefinitionsmodelle durch einen *WS-BPEL-Transformator* um weitere Aktivitäten und Kontrollstrukturen vollautomatisch erweitert. Diese Aktivitäten werden in abgeleiteten Prozessinstanzmodellen innerhalb des Prozessmodelleditors vor dem Prozessbeteiligten verborgen. Sie dienen jedoch zur Prozesslaufzeit dazu, dass eine *Dynamik-Komponente* den Kontrollfluss in Prozessinstanzen innerhalb des Process Servers so beeinflussen kann, dass zusätzliche Prozessfragmente in externen Prozessinstanzen ausgeführt werden, die Ausführung ursprünglich als mandatorisch modellierte Aktivitäten unterbleibt oder bestimmte, bereits (fehlerhaft) durchgeführte Prozessteile wiederholt werden. Hierdurch kann einem Prozessbeteiligten die dynamische Einfügung bzw. Löschung von Aktivitäten bzw. das dynamische Rückspringen innerhalb eines Prozesses *simuliert* werden. Innerhalb der Prozessinstanzmodelle des Prozessmodelleditors werden die indirekten Änderungen im Process Server auf direkte, strukturelle Änderungen in der Ablaufstruktur abgebildet.

Der WS-BPEL-Transformator und die Dynamik-Komponente wurden in XSLT bzw. Java implementiert. Eine GROOVE-Graphersetzungsregelmenge $\mathcal{R}_{trans}$ wurde parallel entwickelt, um die Auswirkungen der Modelltransformationen auf einfache Weise in aus Prozessinstanzmodellen abgeleiteten Prozessgraphen zu erproben, deren Ausführung über die Regelmenge $\mathcal{R}_{bpel}$ simuliert werden kann.

6.1.3 Prüfungen gegen explizites Prozesswissen

Die Erweiterung des Process Servers um die Dynamik-Schicht ermöglicht dynamische Prozessänderungen. Ein im Prozessmodelleditor implementiertes *Prüfwerkzeug* unterstützt Prozessbeteiligte darin, nur *sinnvolle Modelländerungen* vorzunehmen, insbesondere in Prozessinstanzmodellen. Modelländerungen unterliegen Rahmenbedingungen, deren Verletzungen in technischen oder fachlichen Problemen während der Prozessdurchführung resultieren können.

Unter *technische Probleme* in Prozessinstanzmodellen fallen dabei beispielsweise solche Änderungen in der Ablaufstruktur, die zur Verwendung nicht-initialisierter Prozessvariablen oder ungewollter Unerreichbarkeit von Aktivitäten führen können. In Prozesswissensmodellen kann eine ungeschickte Modellierung dazu führen, dass einander widersprechende Reihenfolgebedingungen entstehen. Derartige Bedingungen werden als syntaktische Randbedingungen in der *Object Constraint Language (OCL)* in expliziten Invarianten formuliert, die auf das Metamodell der Prozessmodelle Bezug nehmen. Ein Prüfwerkzeug wertet jeweils anwendbare OCL-Invarianten in den Prozessmodellen aus und gibt dem Prozessbeteiligten *detailliert Rückmeldung* im Falle eines Fehlers. Die Überprüfung technischer Rahmenbedingungen bezieht sich immer nur auf ein Modell und stellt die An- bzw. Abwesenheit der inneren Korrektheit des Modells fest.

Fachliche Rahmenbedingungen sind in Prozesswissensmodellen explizit festgehalten. Prozessinstanz- und -definitionsmodelle können diese fachlichen Bedingungen verletzen, worunter auch die Verletzung der *Komplianz* eines Prozessinstanz- bzw. -definitionsmodells verstanden wird. Zur Feststellung von Komplianzverletzungen nutzt das Prüfwerkzeug aus, dass Prozessinstanz- und -definitionsmodelle dank des gemeinsamen Metamodells syntaktisch mit dem jeweiligen Prozesswissensmodell über Aktivitätstypen verbunden sind und somit jeweils ein *komplexes, syntaktisches Konstrukt* bilden. Bestimmte, unveränderliche OCL-Invarianten nutzen dies aus, indem sie Aussagen über dieses komplexe Konstrukt formulieren. Verletzungen der Invarianten werden dabei soweit möglich so dargestellt, dass das jeweils *verursachende Modellelement* im Prozessinstanz- bzw. -definitionsmodell *markiert* wird und eine *Fehlerbeschreibung* angezeigt wird.

6.1.4 Prüfungen gegen implizites Prozesswissen

Explizites, fachliches Prozesswissen muss in Prozesswissensmodellen zusätzlich modelliert werden. Der hierfür nötige Mehraufwand kann dazu führen, dass Prozesswissensmodelle nur veraltetes und unvollständiges Prozesswissen modellieren. Prüfungen gegen Prozesswissensmodelle werde daher ergänzt um Prüfungen gegen *implizites Wissen* über Prozessabläufe, dass sich *in Prozessinstanz- und -definitionsmodellen* findet. Hierbei kann im Prozessmodelleditor ein Prozessinstanz- oder -definitionsmodell als Prüfling mit einer Menge von anderen Prozessinstanz- oder -definitionsmodellen (Prüfer) paarweise *verglichen* werden. *Unterschiedlich* modellierte, mögliche *Abläufe* werden aufgedeckt und in Form eines *Konsistenzberichts* dem Prozessbeteiligten präsentiert.

Die *Realisierung* des Prozessmodellvergleichs fußt auf einer *Graphgrammatik* mit Regelmenge $\mathcal{R}_{sim}$, die zwei Prozessinstanzmodelle unter Berücksichtigung ihrer im Normalfall vorhandenen Prozesshistorien soweit möglich *simultan ausführt*. Ein simultaner Ausführungsschritt bedeutet dabei die gleichzeitige Finalisierung von zwei gleichgetypten Aktivitäten in unterschiedlichen Prozessinstanzmodellen. Unterschiede in den möglichen Ausführungen erzwingen *nicht-simultane* Schritte, bei denen jeweils nur eine Aktivität finalisiert wird. Solchen nicht-simultanen Schritten werden *Kosten* zugeordnet. Aus dem erzeugten Graphtransitionssystem werden die bzgl. dieser Kosten günstigen Pfade entnommen. Die kumulierten Kosten dieser Pfade liefern dabei ein *quantitatives Maß* für die *Ähnlichkeit* der Prozessmodelle, das zur Ermittlung einander ähnlicher Prozessinstanzmodelle verwendet wird. Zu einem Prüfling ähnliche Prozessinstanzmodelle werden *detaillierten Analysen* unterzogen. Hierbei wird den günstigsten Pfaden entnommen, welche Unterschiede hinsichtlich *Aktivitätshäufigkeit* und *Reihenfolge* zwischen den Modellen bei der Ausführung günstigstenfalls auftreten. Diese Informationen bilden die Einträge im besagten Konsistenzbericht.

6.2 Ausblick

Die Arbeit bietet Raum für einige *Erweiterungen*. Die Wichtigsten werden in diesem Abschnitt diskutiert.

Die erarbeiteten Konzepte, Werkzeuge und Erweiterungen zielen im Gegensatz zu Vorarbeiten im AHEAD-Projekt auf die Einsetzbarkeit in einem industriellen Umfeld ab. Daher wurden bestehende Systeme wie der IBM WebSphere Process Server und standardisierte, verbreitete Sprachen wie WS-BPEL verwendet. Derzeit haben die entwickelten Werkzeuge noch Prototypencharakter. Ein *Überführung* in ein *Produktivsystem* steht noch aus.

Prozesswissensmodelle in BPCL, wie sie in Kapitel 2 beschrieben wurden, formulieren nur Aussagen über Prozessabläufe. Fachliche Bedingungen, die das *Zusammenspiel* zwischen *Aktivitäten*, *Daten* (Produkten) und ausführenden *Ressourcen* betreffen, können darin nicht getroffen werden bzw. wurden nur oberflächlich am Beispiel eines Vier-Augen-Reviews untersucht, bei dem zwei Review-Aktivitäten von jeweils unterschiedlichen Prozessbeteiligten durchgeführt werden müssen [Kur08]. Auch die Sprache *SimBPEL* für Prozessdefinitionsmodelle ist nur eine *vereinfachte Variante* des Sprachstandards WS-BPEL, in der beispielsweise Elemente wie `Pick` fehlen, die durch andere WS-BPEL-Konstrukte nachgebildet werden können. Eine *Angleichung* würde der Einheitlichkeit des Ansatzes zugute kommen.

Die unterstützten *Dynamikmuster*, die in Kapitel 3 beschrieben wurden, decken die unbedingt notwendigen dynamischen Änderungsoperationen für einen Sachbearbeiter ab. Auch hier bleibt Raum für Erweiterungen, beispielsweise *paralleles dynamisches Einfügen*, was jedoch nur in eingeschränktem Maße mit dem Ansatz umzusetzen ist (vgl. Diskussion in Unterabschnitt 3.7.3).

Korrektheitsprüfungen können um weitere Kriterien ergänzt werden, beispielsweise um die Erkennung von Situationen, in denen in parallelen Ausführungspfaden schreibend auf die gleiche Prozessvariable zugriffen wird, was mit dem im Datenbankbereich bekannten “Lost Update”-Problem verwandt ist.

Bei *Konsistenzprüfungen* liegen die derzeitigen Schwächen in der Ergebnisdarstellung. Ein geeignet *verdichteter Konsistenzbericht* ist bei einer großen Menge von Prüfern sicherlich noch hilfreicher als der derzeit implementierte. Auch das Optimierungspotential der Regelmenge $\mathcal{R}_{sim}$ ist insoweit noch nicht ausgeschöpft, als dass immer noch zu viele *unnötige ungünstige Pfade* bei der Transitionssystemerzeugung generiert werden. Abhilfe könnte ein direkter Eingriff in die GROOVE-Implementierung schaffen, der in dieser Arbeit aus Aufwandsgründen unterblieb.

Literaturverzeichnis

[AAA+07] Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri und Alex Yiu: *Web Services Business Process Execution Language v2.0*. Technischer Bericht, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[Aal97] Wil M. P. van der Aalst: *Verification of Workflow Nets*. In: Pierre Azéma und Gianfranco Balbo (Herausgeber): *Application and Theory of Petri Nets 1997*, Band 1248 der Reihe *Lecture Notes in Computer Science*, Seiten 407–426. Springer-Verlag, Berlin, 1997.

[Aal98] Wil M. P. van der Aalst: *The Application of Petri Nets to Workflow Management*. The Journal of Circuits, Systems and Computers, 8(1):21–66, 1998.

[AB02] Wil M. P. van der Aalst und Twan Basten: *Inheritance of workflows: an approach to tackling problems related to change*. Theoretical Computer Science, 270(1–2):125–203, 2002.

[ABHK00] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede und Bartek Kiepuszewski: *Advanced Workflow Patterns*. In: *Conference on Cooperative Information Systems*, Seiten 18–29, 2000.

[AD07] Colin Atkinson und Peter Dadam: *AristaFlow: Komponentenbasierte Anwendungsentwicklung, Prozesskomposition mittels Plug & Play und adaptive Prozessausführung*. In: *Proceedings of the doIT-Forschungstag 2007*, 2007.

[Ada07] Michael Adams: *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. Dissertation, Queensland University of Technology, Mai 2007.

[ADH+03] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm und A. J. M. M. Weijters: *Workflow mining: A survey of issues and approaches*. Data Knowl. Eng., 47(2):237–267, 2003.

[AFL99] Jim Alves-Foss und Fong Shing Lam: *Dynamic Denotational Semantics of Java*. In: Jim Alves-Foss (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *Lecture Notes in Computer Science*, Seiten 201–240. Springer, 1999.

[AH05] Wil M. P. van der Aalst und Arthur T. Hofstede: *YAWL: Yet Another Workflow Language*. Information Systems, 30(4):245–275, 2005.

[AHAE07] Michael Adams, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst und David Edmond: *Dynamic, Extensible and Context-Aware Exception Handling for Workflows*. In: Robert Meersman und Zahir Tari (Herausgeber): *Proceedings of the OTM Conferences 2007 - Volume 1*, Band 4803 der Reihe *Lecture Notes in Computer Science*, Seiten 95–112. Springer, 2007.

[AHEA06] Michael Adams, Arthur H. M. ter Hofstede, David Edmond und Wil M. P. van der Aalst: *Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows*. In: Robert Meersman und Zahir Tari (Herausgeber): *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS 06)*, Nummer 4275 in *Lecture Notes in Computer Science*, Montpellier, Frankreich, November 2006. Springer.

[All05] Thomas Allweyer: *Geschäftsprozessmanagement - Strategie, Entwurf, Implementierung, Controlling*. W3l Verlag, Mai 2005.

[AMW06] Wil M. P. van der Aalst, Ana K. A. de Medeiros und A. J. M. M. Weijters: *Process Equivalence: Comparing Two Process Models Based on Observed Behavior*. In: Schahram Dustdar, José Luiz Fiadeiro und Amit P. Sheth (Herausgeber): *Business Process Management*, Band 4102 der Reihe *Lecture Notes in Computer Science*, Seiten 129–144. Springer, 2006.

[AP06] Wil M. P. van der Aalst und Maja Pesic: *DecSerFlow: Towards a Truly Declarative Service Flow Language*. In: Mario Bravetti, Manuel Núñez und Gianluigi Zavattaro (Herausgeber): *Proceedings of the 3rd International Workshop on Web Services and Formal Method (WS-FM)*, Band 4184 der Reihe *Lecture Notes in Computer Science*, Seiten 1–23. Springer, 2006.

[AtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski und Alistair P. Barros: *Workflow Patterns*. Distrib. Parallel Databases, 14(1):5–51, 2003.

[AW05] Wil M. P. van der Aalst und Mathias Weske: *Case Handling: A New Paradigm for Business Process Support*. Data Knowl. Eng., 53(2):129–162, 2005.

[Bal98] Helmut Balzert: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag GmbH, Heidelberg, 1998.

[Bau99] Roland Baumann: *Ein Datenbankmanagementsystem für verteilte, integrierte Softwareentwicklungsumgebungen*. Dissertation, RWTH Aachen, 1999.

[BBBE07] Michael Beisiegel, Henning Blohm, Dave Booz und Mike Edwards: *SCA Assembly Model V1.00*. Open SOA, März 2007. `http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[BC92] Luca Bernardinello und Fiorella de Cindio: *A survey of basic net models and modular net classes*. In: Grzegorz Rozenberg (Herausgeber): *Advances in Petri Nets: The DEMON Project*, Band 609 der Reihe *Lecture Notes in Computer Science*, Seiten 304–351. Springer, 1992.

[Bec01] Simon M. Becker: *Integration von Werkzeugen für das Management übergreifender Prozesse*. Diplomarbeit, RWTH Aachen, 2001.

[Bec07] Simon M. Becker: *Integratoren zu Konsistenzsicherung von Dokumenten in Entwicklungsprozessen*. Dissertation, RWTH Aachen, 2007.

[BGS07] Domenico Bianculli, Carlo Ghezzi und Paola Spoletini: *A Model Checking Approach to Verify BPEL4WS Workflows*. In: *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications 2007*, Seiten 13–20. IEEE Computer Society, 2007.

[Böh06] Boris Böhlen: *Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge*. Dissertation, RWTH Aachen, Aachen, 2006. 250 pp.

[BJSW02] Boris Böhlen, Dirk Jäger, Ansgar Schleicher und Bernhard Westfechtel: *UPGRADE: A Framework for Building Graph-Based Interactive Tools.* Electronic Notes in Theoretical Computer Science, 72:2:113–123, 2002.

[Bra85] Wilfried Brauer (Herausgeber): *Proceedings of the 12th Colloquium Automata, Languages and Programming*, Band 194 der Reihe *Lecture Notes in Computer Science*. Springer, 1985.

[BSMP09] Frank Budinsky, Dave Steinberg, Ed Merks und Marcelo Paternostro: *Eclipse Modeling Framework.* The Eclipse Series. Addison-Wesley Professional, 2. Auflage, 2009.

[Bun97] Horst Bunke: *On a relation between graph edit distance and maximum common subgraph.* Pattern Recognition Letters, 18(8):689–694, 1997.

[BY92] Ricardo A. Baeza-Yates: *Text-Retrieval: Theory and Practice.* In: Jan van Leeuwen (Herausgeber): *IFIP Congress (1)*, Band A-12 der Reihe *IFIP Transactions*, Seiten 465–476. North-Holland, 1992.

[Cas98] Fabio Casati: *Models, Semantics, and Formal Methods for the Design of Workflows and their Exceptions.* Dissertation, Politecnico di Milano, 1998.

[CCPP99] Fabio Casati, Stefano Ceri, Stefano Paraboschi und Guiseppe Pozzi: *Specification and Implementation of Exceptions in Workflow Management Systems.* ACM Trans. Database Syst., 24(3):405–451, 1999.

[CCPP00] Fabio Casati, Stefano Ceri, Barbara Pernici und Giuseppe Pozzi: *Conceptual Modeling of Workflows.* In: *Advances in Object-Oriented Data Modeling*, Seiten 281–306. 2000.

[CG70] Derek G. Corneil. und Calvin C. Gotlieb: *An Efficient Algorithm for Graph Isomorphism.* Journal of the ACM, 17(1):51–64, 1970.

[CG85] Stefano Ceri und Georg Gottlob: *Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries.* IEEE Trans. Softw. Eng., 11(4):324–345, 1985.

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

[CM84] K. M. Chandy und J. Misra: *The Drinking Philosophers Problem*. ACM Trans. on Programming Languages and Sys., 6(4):632, Oktober 1984.

[COS] *COSA GmbH*. `http://www.ley.de`.

[CR09] Erich Clayberg und Dan Rubel: *Eclipse Plug-ins*. The Eclipse Series. Addison-Wesley Professional, 3. Auflage, 2009.

[CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick und C. Michael Pilato: *Version Control with Subversion*. O'Reilly & Associates, Sebastopol, Juni 2004.

[DAH05] Marlon Dumas, Wil M. P. van der Aalst und Arthur H. M. ter Hofstede (Herausgeber): *Process-Aware Information Systems*. John Wiley & Sons, 2005.

[DAV05] Boudewijn F. van Dongen, Wil M. P. van der Aalst und Henricus M. W. Verbeek: *Verification of EPCs: Using Reduction Rules and Petri Nets*. In: Oscar Pastor und Joao Falcao e Cunha [PC05], Seiten 372–386.

[DBL03] *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*. ACM, 2003.

[DDO07] Remco M. Dijkman, Marlon Dumas und Chun Ouyang: *Formal Semantics and Analysis of BPMN Process Models*. Technischer Bericht, Queensland University of Technology, 2007.

[Dij59] Edsger W. Dijkstra: *A note on two problems in connexion with graphs*. Numerische Mathematik, 1:269–271, 1959.

[Dij07] Remco M. Dijkman: *A Classification of Differences between Similar Business Processes*. In: *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, Seite 37, Washington, DC, USA, 2007. IEEE Computer Society.

[Dij08] Remco M. Dijkman: *Diagnosing Differences between Business Process Models*. In: Marlon Dumas et al. [DRS08], Seiten 261–277.

[DR09] Peter Dadam und Manfred Reichert: *The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements.* Computer Science - Research and Development, 23(2):81–97, 2009.

[DRS08] Marlon Dumas, Manfred Reichert und Ming-Chien Shan (Herausgeber): *Proceedings of the 6th International Conference on Business Process Management*, Band 5240 der Reihe *Lecture Notes in Computer Science*. Springer, 2008.

[EHHS00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel und Stefan Sauer: *Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML.* In: Andy Evans, Stuart Kent und Bran Selic (Herausgeber): *UML*, Band 1939 der Reihe *Lecture Notes in Computer Science*, Seiten 323–337. Springer, 2000.

[Ehs08] Nicolas Ehses: *A-posteriori-Erweiterung eines Workflow-Management-Systems um die Ausführung dynamischer Workflows.* Diplomarbeit, RWTH Aachen, Januar 2008.

[EKL+08] Markus Eggersmann, Bernhard Kausch, Holger Luczak, Wolfgang Marquardt, Christopher M. Schlick, Nicole Schneider, Ralph Schneider und Manfred Theißen: *Work Process Models.* In: Manfred Nagl und Wolfgang Marquardt (Herausgeber): *Results of the IMPROVE Project*, Nummer 4970 in *Lecture Notes in Computer Science*, Seiten 126–152. Springer, 2008.

[FESS07] Alexander Förster, Gregor Engels, Tim Schattkowsky und Ragnhild Van Der Straeten: *Verification of Business Process Quality Constraints Based on Visual Process Patterns.* In: *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, Seiten 197–208. IEEE Computer Society, 2007.

[Fis09] André Fischer: *Werkzeuggestützte Auswertung impliziten Prozesswissens.* Diplomarbeit, RWTH Aachen, 2009.

[FMRS07] Christian Fuss, Christof Mosler, Ulrike Ranger und Erhard Schultchen: *The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES.* In: *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*, 2007. 14 pp.

[För08] Alexander Förster: *Pattern-Based Business Process Design and Verification*. Dissertation, Universität Paderborn, August 2008.

[FW97] Frank von Fürstenwerth und Alfons Weiß: *Versicherungsalphabet*. Verlag Versicherungswirtschaft e.V., 1997.

[GAJVR08] Florian Gottschalk, Wil M. P. van der Aalst, Monique H. Jansen-Vullers und Marcello La Rosa: *Configurable Workflow Models*. Int. J. Cooperative Inf. Syst., 17(2):177–221, 2008.

[GJS05] James Gosling, Bill Joy und Guy Steele: *The Java^{TM} Language Specification*. Sun Microsystems, 3. Auflage, Mai 2005. `http://java.sun.com/doc/language_specification.html`.

[GK01] Martin Gogolla und Cris Kobryn (Herausgeber): *Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Band 2185 der Reihe *Lecture Notes in Computer Science*. Springer, 2001.

[GKP98] Robert Geisler, Marcus Klar und Claudia Pons: *Dimensions and Dichotomy in Metamodeling*. In: *In Proceedings of the Third BCS-FACS Northern Formal Methods Workshop*. Springer, 1998. `mack.ittc.ku.edu/geisler98dimensions.html`.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler und Steven Völkel: *MontiCore: a framework for the development of textual domain specific languages*. In: Wilhelm Schäfer, Matthew B. Dwyer und Volker Gruhn (Herausgeber): *ICSE Companion*, Seiten 925–926. ACM, 2008.

[Gla90] Rob J. van Glabbeek: *The linear time-branching time spectrum (extended abstract)*. In: *Proceedings on Theories of concurrency : unification and extension (CONCUR)*, Seiten 278–297, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Gro03] Object Management Group: *Common Warehouse Metamodel v1.1*, 2003. `http://www.omg.org/cgi-bin/doc?formal/03-03-02`.

[Gro05] Object Management Group: *OMG Unified Modeling Language 2.1.2*. OMG, http://www.omg.com/uml/, 2005.

[Gro09] Richard C. Gronback: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Pearson Education, 2009.

[Han97] Yanbo Han: *HOON - A Formalism Supporting Adaptive Workflows.* Technischer Bericht, Department of Computer Science, University of Georgia, 1997.

[Har87] D. Harel: *Statecharts: A visual formalism for complex systems.* Science of Computer Programming, 8(3):231–274, Juni 1987.

[Hau05] Jan Hendrik Hausmann: *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages.* Dissertation, University of Paderborn, 2005.

[Hav05] Michael Havey: *Essential Business Process Modeling.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, August 2005.

[HAW09] Thomas Heer, Christoph Außem und René Wörzberger: *Flexible Multi-Dimensional Visualization of Process Enactment Data.* In: *Business Process Management Workshops*, Lecture Notes in Business Information Processing. Springer, 2009. (erscheint 2010).

[HBR08] Alena Hallerbach, Thomas Bauer und Manfred Reichert: *Managing Process Variants in the Process Life Cycle.* In: José Cordeiro und Joaquim Filipe (Herausgeber): *ICEIS (3-2)*, Seiten 154–161, 2008.

[HBW08] Thomas Heer, Christoph Briem und René Wörzberger: *Workflows in Dynamic Development Processes.* In: Danilo Ardagna, Massimo Mecella und Jian Yang (Herausgeber): *Business Process Management Workshops*, Band 17 der Reihe *Lecture Notes in Business Information Processing*, Seiten 266–277. Springer, 2008.

[Hel08] Markus Heller: *Dezentralisiertes, sichtenbasiertes Management übergreifender Entwicklungsprozesse.* Dissertation, RWTH Aachen, Aachen, 2008. 501 pp.

[HHH+08] Ri Hai, Thomas Heer, Markus Heller, Manfred Nagl, Ralph Schneider, Bernhard Westfechtel und René Wörzberger: *Administration Models and Management Tools.* In: Manfred Nagl und Wolfgang Marquardt (Herausgeber): *Results of the IMPROVE Project*, Nummer 4970 in *Lecture Notes in Computer Science*, Seiten 621–628. 2008.

[HHJS97] Jens Hagemeyer, Thomas Herrmann, Katharina Just und Rüdiger Striemer: *Flexibilität bei Workflow-Management-Systemen*. In: *Proc. Software-Ergonomie*, Seiten 179–190. Teubner, März 1997.

[HHM+06] Ri Hai, Markus Heller, Wolfgang Marquardt, Manfred Nagl und René Wörzberger: *Workflow Support for Inter-organizational Design Processes*. In: *Proceedings of the 6th European Symposium on Computer Aided Process Engineering and 9th International Symposium on Process Systems Engineering*, Seiten 2027–2032. Elsevier, 2006.

[HHR04] Lutz J. Heinrich, Armin Heinzl und Friedrich Roithmayr: *Wirtschaftsinformatik-Lexikon*. Oldenburg Verlag, 7. Auflage, 2004.

[HHS04] Jan Hendrik Hausmann, Reiko Heckel und Stefan Sauer: *Dynamic Meta Modeling with time: Specifying the semantics of multimedia sequence diagrams*. Software and System Modeling, 3(3):181–193, 2004.

[HJH96] Thomas Hermann und Katharina Just-Hahn: *Organizational learning with flexible workflow management systems*. SIGOIS Bull., 17(3):54–57, 1996.

[HJK+08] Markus Heller, Dirk Jäger, Carl-Arndt Krapp, Manfred Nagl, Ansgar Schleicher, Bernhard Westfechtel und René Wörzberger: *An Adaptive and Reactive Management System for Project Coordination*. In: Manfred Nagl und Wolfgang Marquardt (Herausgeber): *Results of the IMPROVE Project*, Nummer 4970 in *Lecture Notes in Computer Science*, Seiten 300–366. 2008.

[HJKW96] Peter Heimann, Gregor Joeris, Carl-Arndt Krapp und Bernhard Westfechtel: *DYNAMITE: Dynamic task nets for software process management*. In: *Proceedings of the 18th International Conference on Software Engineering*, Seiten 331–341. IEEE Computer Society Press, 1996.

[HKWJ97] Peter Heimann, Carl-Arndt Krapp, Bernhard Westfechtel und Gregor Joeris: *Graph-Based Software Process Management*. International Journal of Software Engineering and Knowledge Engineering, 7(4):431–455, 1997.

[HMU02] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexi-*

tätstheorie. Pearson Studium, München, 2002. 2., überarbeitete Auflage.

[Hoa85] Charles A. R. Hoare: *Communicating Sequential Processes*. Prentice Hall, 1985.

[HR04] David Harel und Bernhard Rumpe: *Meaningful Modeling: What's the Semantics of Semantics?* IEEE Computer, 37(10):64–72, 2004.

[HSW03] Markus Heller, Ansgar Schleicher und Bernhard Westfechtel: *Process Evolution Support in the AHEAD System*. In: John L. Pfaltz et al. [PNB04], Seiten 454–460.

[HW06a] Markus Heller und René Wörzberger: *A Management System Supporting Interorganizational Cooperative Development Processes in Chemical Engineering*. In: *9th World Conference on Integrated Design and Process Technology (IDPT 2006), June 25, 2006 – June 30, 2006, San Diego, California, USA*, Seiten 639–650. SDPS, 2006.

[HW06b] Markus Heller und René Wörzberger: *Management Support of Interorganizational Cooperative Software Development Processes based on Dynamic Process Views*. In: *15th International Conference on Software Engineering and Data Engineering (SEDE 2006), July 6–8, 2006, Los Angeles, California, USA, 7 pages*, Seiten 15–28, 2006.

[HW07] Markus Heller und René Wörzberger: *A Management System Supporting Interorganizational Cooperative Development Processes in Chemical Engineering*. Journal of Integrated Design and Process Science: Transactions of the SDPS, 10(2):57–78, 2007.

[HW09] Thomas Heer und René Wörzberger: *Support for Modeling, Enactment and Monitoring of Engineering Design Processes*. In: *Proceedings of the 8th World Congress of Chemical Engineering (WCCE8)*, 2009.

[Int96] International Organization for Standardization: *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, 1996. `http://www.iso.ch/cate/d26153.html`.

[Int03] International Organization for Standardization: *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003. `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110`.

[JB96] Stefan Jablonski und Christoph Bussler: *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.

[JBS99] Stefan Jablonski, Markus Böhm und Wolfgang Schulze: *Workflow-Management: Entwicklung von Anwendungen und Systemen*. dpunkt, 1999.

[Jäg03] Dirk Jäger: *Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen*. Dissertation, RWTH Aachen, 2003.

[Joe00] Gregor Joeris: *Flexibles und adaptives Workflowmanagement für verteilte und dynamische Prozesse*. Dissertation, Universität Bremen, 2000.

[JSBW00] Dirk Jäger, Ansgar Schleicher und B. Bernhard Westfechtel: *AHEAD: A Graph-Based System for Modeling and Managing Development Processes*. In: Manfred Nagl, Andy Schürr und Manfred Münch (Herausgeber): *Applications of Graph Transformations with Industrial Relevance: International Workshop, AGTIVE'99, Kerkrade, The Netherlands, September 1999. Proceedings*, Band 1779 der Reihe *LNCS*, Seiten 325–340. Springer, 2000.

[KER99] Stuart Kent, Andy Evans und Bernhard Rumpe: *UML Semantics FAQ*. In: Ana M. D. Moreira und Serge Demeyer (Herausgeber): *ECOOP Workshops*, Band 1743 der Reihe *Lecture Notes in Computer Science*. Springer, December 1999. `http://www.cs.kent.ac.uk/pubs/1999/977`.

[KGFE08] Jochen M. Küster, Christian Gerth, Alexander Förster und Gregor Engels: *Detecting and Resolving Process Model Differences in the Absence of a Change Log*. In: Marlon Dumas et al. [DRS08], Seiten 244–260.

[KK02] Dimitris Karagiannis und Harald Kühn: *Metamodelling Platforms*. In: Kurt Bauknecht, A. Min Tjoa und Gerald Quirchmayr (Herausgeber): *EC-Web*, Band 2455 der Reihe *Lecture Notes in Computer Science*, Seite 182. Springer, 2002.

[KKSW98] Carl-Arndt Krapp, Sven Krüppel, Ansgar Schleicher und Bernhard Westfechtel: *Graph-Based Models for Managing Development Processes, Resources, and Products*. In: *Proceedings of the 6th International Workshop on Theory and Appplications of Graph Transformations (TAGT 1998)*, Band 1764 der Reihe *LNCS*, Seiten 455–474. Springer, November 1998.

[KNS92] G. Keller, M. Nüttgens und A. W. Scheer: *Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK)*. Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.

[Kra98] Carl-Arndt Krapp: *An Adaptable Environment for the Management of Development Processes*. Dissertation, RWTH Aachen, 1998.

[KRV07] Holger Krahn, Bernhard Rumpe und Steven Völkel: *Integrated Definition of Abstract and Concrete Syntax for Textual Languages*. In: Gregor Engels, Bill Opdyke, Douglas C. Schmidt und Frank Weil (Herausgeber): *MoDELS*, Band 4735 der Reihe *Lecture Notes in Computer Science*, Seiten 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe und Steven Völkel: *MontiCore: Modular Development of Textual Domain Specific Languages*. In: Richard F. Paige und Bertrand Meyer (Herausgeber): *TOOLS (46)*, Band 11 der Reihe *Lecture Notes in Business Information Processing*, Seiten 297–315. Springer, 2008.

[KS02] Friedrich Kluge und Elmar Seebold: *Etymologisches Wörterbuch der deutschen Sprache*. Gruyter, 24. Auflage, 2002.

[KSW95] Norbert Kiesel, Andy Schürr und Bernhard Westfechtel: *GRAS, a Graph-Oriented (Software) Engineering Database System*. Inf. Syst., 20(1):21–51, 1995.

[Kur08] Thomas Kurpick: *Werkzeuggestützte Prüfungen dynamischer Prozesse*. Diplomarbeit, RWTH Aachen, Dezember 2008.

[KW97] Carl-Arndt Krapp und Bernhard Westfechtel: *Feedback Handling in Dynamic Task Nets*. In: *Proceedings of the 12th International Conference on Automated Software Engineering, Incline Village,* *Nevada, USA*, Seiten 301–302. IEEE Computer Society Press, 1997.

[Lev66] Vladimir I. Levenshtein: *Binary codes capable of correcting deletions, insertions and reversals.* Soviet Physics Doklady., 10(8):707–710, Februar 1966.

[LGRMD08] Linh Thao Ly, Kevin Göser, Stefanie Rinderle-Ma und Peter Dadam: *Compliance of Semantic Constraints - A Requirements Analysis for Process Management Systems.* In: *Proc. 1st Int'l Workshop on Governance, Risk and Compliance - Applications in Information Systems (GRCIS'08)*, 2008.

[LM07] Roberto Lucchi und Manuel Mazzara: *A pi-calculus based semantics for WS-BPEL.* J. Log. Algebr. Program, 70(1):96–118, 2007.

[LRD06] Linh Thao Ly, Stefanie Rinderle und Peter Dadam: *Semantic Correctness in Adaptive Process Management Systems.* In: Schahram Dustdar, José Luiz Fiadeiro und Amit P. Sheth (Herausgeber): *Business Process Management*, Band 4102 der Reihe *Lecture Notes in Computer Science*, Seiten 193–208. Springer, 2006.

[LRD08] Linh Thao Ly, Stefanie Rinderle und Peter Dadam: *Integration and verification of semantic constraints in adaptive process management systems.* Data Knowl. Eng., 64(1):3–23, 2008.

[LRW08a] Chen Li, Manfred Reichert und Andreas Wombacher: *Discovering Reference Process Models by Mining Process Variants.* In: *ICWS*, Seiten 45–53. IEEE Computer Society, 2008.

[LRW08b] Chen Li, Manfred Reichert und Andreas Wombacher: *On Measuring Process Model Similarity Based on High-Level Change Operations.* In: Qing Li, Stefano Spaccapietra, Eric Yu und Antoni Olivé (Herausgeber): *ER*, Band 5231 der Reihe *Lecture Notes in Computer Science*, Seiten 248–264. Springer, 2008.

[MA06] Jan Mendling und Wil M. P. van der Aalst: *Towards EPC Semantics based on State and Context.* In: Markus Nüttgens, Frank J. Rump und Jan Mendling (Herausgeber): *EPK*, Band 224 der Reihe *CEUR Workshop Proceedings*, Seiten 25–48. CEUR-WS.org, 2006.

[MA07] Jan Mendling und Wil M. P. van der Aalst: *Formalization and Verification of EPCs with OR-Joins Based on State and Context*. In: John Krogstie, Andreas L. Opdahl und Guttorm Sindre (Herausgeber): *CAiSE*, Band 4495 der Reihe *Lecture Notes in Computer Science*, Seiten 439–453. Springer, 2007.

[Mac77] Lutz Mackensen: *Großes Deutsches Wörterbuch. Rechtschreibung, Grammatik, Stil, Worterklärung, Fremdwörterbuch*. Kapp Verlag, 1977.

[MAW08] Ana Karla Alves de Medeiros, Wil M. P. van der Aalst und A. J. M. M. Weijters: *Quantifying process equivalence based on observed behavior*. Data Knowl. Eng., 64(1):55–74, 2008.

[MB98] Bruno T. Messmer und Horst Bunke: *A New Algorithm for Error-Tolerant Subgraph Isomorphism Detection*. IEEE Trans. Pattern Anal. Mach. Intell., 20(5):493–504, 1998.

[MB99] Bruno T. Messmer und Horst Bunke: *A decision tree approach to graph and subgraph isomorphism detection*. Pattern Recognition, 32(12):1979–1998, 1999.

[Men07] Jan Mendling: *Detection and Prediction of Errors in EPC Business Process Models*. Dissertation, Vienna University of Economics and Business Administration, Mai 2007.

[Mil80] Robin Milner: *A Calculus of Communicating Systems*, Band 92 der Reihe *Lecture Notes in Computer Science*. Springer, 1980.

[MMN06] Jan Mendling, Michael Moser und Gustaf Neumann: *Transformation of yEPC business process models to YAWL*. In: Hisham Haddad (Herausgeber): *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, Frankreich, April, 2006*, Seiten 1262–1266. ACM, 2006.

[MS97] Alexandru Mateescu und Arto Salomaa: *Formal languages: an introduction and a synopsis*. Handbook of formal languages, vol. 1: word, language, grammar, 1:1–39, 1997.

[MS03] Peter J. Mangan und Shazia W. Sadiq: *A Constraint Specification Aproach to Building Flexible Workflows*. Journal of Research and Practice in Information Technology, 35(1):21–39, 2003.

[Nag79] Manfred Nagl: *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979.

[Nag96] Manfred Nagl (Herausgeber): *Building tightly integrated software development environments: the IPSEN approach*. Springer, New York, NY, USA, 1996.

[NN92] Hanne Riis Nielson und Flemming Nielson: *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992. 240 pages.

[NPS91] Mauro Negri, Giuseppe Pelagatti und Licia Sbattella: *Formal Semantics of SQL Queries*. ACM Trans. Database Syst., 16(3):513–534, 1991.

[NR02] Markus Nüttgens und Frank J. Rump: *Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK)*. In: Jörg Desel und Mathias Weske (Herausgeber): *Promise*, Band 21 der Reihe *LNI*, Seiten 64–77. GI, 2002.

[NW94] Manfred Nagl und Bernhard Westfechtel: *A Universal Component for the Administration in Distributed and Integrated Development Environments*. Technischer Bericht, RWTH Aachen, 1994.

[Obj04] Object Management Group: *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*, 2004. `http://www.omg.org/cgi-bin/doc?ptc/03-10-04`.

[Obj08] Object Management Group: *Business Process Modeling Notation, V1.1*, Januar 2008. `http://www.omg.org/spec/BPMN/1.1/PDF`.

[OVA+07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas und Arthur H. M. ter Hofstede: *Formal semantics and analysis of control flow in WS-BPEL (Revised Version)*. Sci. Comput. Program., 67(2-3):162–198, 2007.

[PA06] Maja Pesic und Wil M. P. van der Aalst: *A Declarative Approach for Flexible Business Processes Management*. In: Johann Eder und Schahram Dustdar (Herausgeber): *Business Process Management Workshops*, Band 4103 der Reihe *Lecture Notes in Computer Science*, Seiten 169–180. Springer, 2006.

[PC05] Oscar Pastor und Joao Falcao e Cunha (Herausgeber): *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, Band 3520 der Reihe *Lecture Notes in Computer Science*. Springer, 2005.

[Pes08] Maja Pesic: *Constraint-Based Workflow Management Systems: Shifting Control to Users*. Dissertation, TU Einhoven, 2008.

[Pet62] Carl-Adam Petri: *Kommunikation mit Automaten*. Dissertation, Universität Bonn, 1962.

[PNB03] John L. Pfaltz, Manfred Nagl und Boris Böhlen (Herausgeber): *VisualDiaGen - A Tool for Visually Specifying and Generating Visual Editors*, Band 3062 der Reihe *Lecture Notes in Computer Science*. Springer, 2003.

[PNB04] John L. Pfaltz, Manfred Nagl und Boris Böhlen (Herausgeber): *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, Band 3062 der Reihe *Lecture Notes in Computer Science*. Springer, 2004.

[Pnu85] Amir Pnueli: *Linear and Branching Structures in the Semantics and Logics of Reactive Systems*. In: Wilfried Brauer [Bra85], Seiten 15–32.

[PSA07] Maja Pesic, Helen Schonenberg und Wil M. P. van der Aalst: *DECLARE: Full Support for Loosely-Structured Processes*. In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), Oktober 2007, Annapolis, Maryland, USA*, Seiten 287–300, 2007.

[RA07] Michael Rosemann und Wil M. P. van der Aalst: *A configurable reference modelling language*. Inf. Syst, 32(1):1–23, 2007.

[RAHE05] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede und David Edmond: *Workflow Resource Patterns: Identification, Representation and Tool Support*. In: Oscar Pastor und Joao Falcao e Cunha [PC05], Seiten 216–232.

[RD98] Manfred Reichert und Peter Dadam: *ADEPTflex-Supporting Dynamic Changes of Workflows Without Loosing Control*. Journal of Intelligent Information Systems, 10(2):93–129, 1998.

[RDH08] Marcello la Rosa, Marlon Dumas und Arthur H. M. ter Hofstede: *Modelling Business Process Variability*. Technischer Bericht, Faculty of Information Technology - Queensland University of Technology, 2008.

[Rei86] Wolfgang Reisig: *Petrinetze. Eine Einführung*. Springer, 1986.

[Rei00] Manfred Reichert: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. Dissertation, Universität Ulm, Mai 2000.

[Ren03] Arend Rensink: *The GROOVE Simulator: A Tool for State Space Generation*. In: John L. Pfaltz et al. [PNB04], Seiten 479–485.

[Ren08] Arend Rensink: *Explicit State Model Checking for Graph Grammars*. In: Pierpaolo Degano, Rocco De Nicola und José Meseguer (Herausgeber): *Concurrency, Graphs and Models*, Band 5065 der Reihe *Lecture Notes in Computer Science*, Seiten 114–132. Springer, 2008.

[RHEA05] N. Russell, A.H.M. ter Hofstede, D. Edmond und W.M.P. van der Aalst: *Workflow Data Patterns: Identification, Representation and Tool Support*. In: Lois M. L. Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos und Oscar Pastor (Herausgeber): *24nd International Conference on Conceptual Modeling (ER 2005)*, Band 3716 der Reihe *Lecture Notes in Computer Science*, Seiten 353–368. Springer, 2005.

[RK09] Arend Rensink und Jan-Hendrik Kuperus: *Repotting the geraniums: on nested graph transformation rules*. In: A. Boronat und R. Heckel (Herausgeber): *Graph transformation and visual modelling techniques, York, U.K.*, Band 18 der Reihe *Electronic Communications of the EASST*. EASST, 2009.

[RR04] Klaus Richter und Jan-Michael Rost: *Komplexe Systeme*. Fischer Verlag, 2. Auflage, Mai 2004.

[RS82] Siegrid Radszuweit und Martha Spalier: *Knaurs Wörterbuch der Synonyme - Der treffende Ausdruck - Das passende Wort*. Lexikographisches Institut München, 1982.

[RT98] Bernhard Rumpe und Veronika Thurner: *Refining Business Processes*. In: Haim Kilov, Bernhard Rumpe und Ian Simmonds (Herausgeber): *Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*. Technical University Munich, TUM-I9820, 1998.

[Rum04a] Bernhard Rumpe: *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2004.

[Rum04b] Bernhard Rumpe: *Modellierung mit UML*. Springer, 2004.

[RW09] A. Reynolds und M. Wright: *Oracle SOA Suite Developer's Guide*. Packt Publishing, März 2009.

[Sch91] Andy Schürr: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991.

[Sch02] Ansgar Schleicher: *Management of Development Processes: An Evolutionary Approach*. Dissertation, RWTH Aachen, 2002.

[SG03] Ralph Schneider und Sascha Gerhards: *WOMS – A Work Process Modeling Tool*. In: M. Nagl und B. Westfechtel (Herausgeber): *Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, Seiten 375–376. Wiley-VHC, 2003.

[sha] *Enhydra-Shark*. `http://www.enhydra.org/workflow/shark`.

[SK95] Kenneth Slonneger und Barry L. Kurtz: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[SMR+08] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar und Wil M. P. van der Aalst: *Process Flexibility: A Survey of Contemporary Approaches*. In: Jan L. G. Dietz, Antonia Albani und Joseph Barjis (Herausgeber): *CIAO! / EOMAS*, Band 10 der Reihe *Lecture Notes in Business Information Processing*, Seiten 16–30. Springer, 2008.

[SO00] Wasim Sadiq und Maria E. Orlowska: *Analyzing Process Models Using Graph Reduction Techniques*. Inf. Syst., 25(2):117–134, 2000.

[SP06] Arnd Schnieders und Frank Puhlmann: *Variability Mechanisms in E-Business Process Families*. In: Witold Abramowicz und Heinrich C. Mayr (Herausgeber): *9th International Conference on Business Information Systems (BIS)*, Band 85 der Reihe *LNI*, Seiten 583–601. GI, 2006.

[Spi89] J. M. Spivey: *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[SSO01] Shazia W. Sadiq, Wasim Sadiq und Maria E. Orlowska: *Pockets of Flexibility in Workflow Specification*. In: Hideko S. Kunii, Sushil Jajodia und Arne Sølvberg (Herausgeber): *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling*, Nummer 2224 in *Lecture Notes in Computer Science*, Seiten 513–526. Springer, 2001.

[Sta73] Herbert Stachowiak: *Allgemeine Modelltheorie.* Springer, 1973.

[SW01] Ansgar Schleicher und Bernhard Westfechtel: *Beyond Stereotyping: Metamodeling for the UML.* In: *Proceedings of the 34^{th} Hawaii International Conference on System Sciences (HICSS), Minitrack: Unified Modeling – A Critical Review and Suggested Future, 10 pages*. IEEE Computer Society Press, 2001.

[SWZ99] Andy Schürr, Andreas Winter und Albert Zündorf: *The PROGRES Approach: Language and Environment.* In: Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski und Gregor Rozenberg (Herausgeber): *Handbook on Graph Grammars and Computing by Graph Transformation – Volume 2: Applications, Languages, and Tools*, Seiten 478–550. World Scientific, 1999.

[Tae03] Gabriele Taentzer: *AGG: A Graph Transformation Environment for Modeling and Validation of Software.* In: John L. Pfaltz et al. [PNB04], Seiten 446–453.

[TSMB95] Stephanie Teufel, Christian Sauter, Thomas Muehlherr und Kurt Bauknecht: *Computerunterstützung für die Gruppenarbeit.* Addison Wesley, Zuerich, 1995.

[VA00] Henricus M. W. Verbeek und Wil M. P. van der Aalst: *Woflan 2.0: A Petri-Net-Based Workflow Diagnosis Tool.* In: Edited by G. Goos, J. Hartmanis und J. van Leeuwen (Herausgeber): *Application and Theory of Petri Nets 2000*, Nummer 1825 in *Lecture Notes in Computer Science*, Seiten 475–484. Springer, 2000.

[VA05] Henricus M. W. Verbeek und Wil M. P. van der Aalst: *Analyzing BPEL Processes using Petri Nets.* In: D. Marinescu (Herausgeber): *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination*, Seiten 59–78, Miami, Florida, USA, 2005. Florida International University.

[VBA01] Henricus M. W. Verbeek, Twan Basten und Wil M. P. van der Aalst: *Diagnosing Workflow Processes using Woflan.* Comput. J., 44(4):246–279, 2001.

[Ver04] Henricus M. W. Verbeek: *Verification of WF-nets.* Dissertation, Technische Universität Eindhoven, 2004.

[VW98] Gottfried Vossen und Matthias Weske: *The WASA Approach to Workflow Management for Scientific Applications.* In: *Workflow*

Management Systems and Interoperability, Band 164, Seiten 145–164. Springer, 1998.

[WAM+07] Ueli Wahli, Vedavyas Avula, Hannah Macleod, Mohamed Saeed und Anders Vinther: *Business Process Management: Modeling through Monitoring Using WebSphere V6.0.2 Products*. IBM, 1. Auflage, August 2007. `http://www.redbooks.ibm.com/redpieces/abstracts/sg247148.html`.

[WEH08] René Wörzberger, Nicolas Ehses und Thomas Heer: *Adding Support for Dynamics Patterns to Static Business Process Management Systems*. In: Cesare Pautasso und Éric Tanter (Herausgeber): *Software Composition*, Band 4954 der Reihe *Lecture Notes in Computer Science*, Seiten 84–91. Springer, 2008.

[Wes99a] Mathias Weske: *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects*. Habilitationsschrift, Westfälische Wilhelms-Universität Münster, 1999.

[Wes99b] Bernhard Westfechtel: *Models and Tools for Managing Development Processes*, Band 1646 der Reihe *Lecture Notes in Computer Science*. Springer, 1999.

[Wes01] B. Westfechtel: *Ein graphbasiertes Managementsystem für dynamische Entwicklungsprozesse*. Informatik Forschung und Entwicklung, 16(3):125–144, 2001.

[WG08] Peter Y. H. Wong und Jeremy Gibbons: *A Process Semantics for BPMN*. In: Shaoying Liu, T. S. E. Maibaum und Keijiro Araki (Herausgeber): *ICFEM*, Band 5256 der Reihe *Lecture Notes in Computer Science*, Seiten 355–374. Springer, 2008.

[WH08] René Wörzberger und Thoams Heer: *Process Model Editing Support Using Eclipse Modeling Project Tools*. In: Peter Friese, Simon Zambrovski und Frank Zimmermann (Herausgeber): *Proceedings of the Second Workshop on MDSD Today*, Lecture Notes in Informatics. Shaker Verlag, 2008.

[WHH07] René Wörzberger, Markus Heller und Frank Walter Häßler: *Evaluating Workflow Definition Language Revisions with Graph-Based Tools*. Electronic Communications of the EASST, 6, 2007.

[WKH08a] René Wörzberger, Thomas Kurpick und Thomas Heer: *Checking Correctness and Compliance of Integrated Process Models*. In:

Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Seiten 576–583, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[WKH08b] René Wörzberger, Thomas Kurpick und Thomas Heer: *On Correctness, Compliance and Consistency of Process Models*. In: *Proceedings of the 17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE)*, Seiten 251–252. IEEE Computer Society, 2008.

[WKR01] Andreas Winter, Bernt Kullbach und Volker Riediger: *An Overview of the GXL Graph Exchange Language*. In: Stephan Diehl (Herausgeber): *Software Visualization*, Band 2269 der Reihe *Lecture Notes in Computer Science*, Seiten 324–336. Springer, 2001.

[Wör04] René Wörzberger: *Methoden und Werkzeuge zur UML-basierten Spezifikation von Integrationsregeln*. Diplomarbeit, RWTH Aachen, 2004.

[WR06] Andreas Wombacher und Maarten Rozie: *Evaluation of Workflow Similarity Measures in Service Discovery*. In: Mareike Schoop, Christian Huemer, Michael Rebstock und Martin Bichler (Herausgeber): *Service Oriented Electronic Commerce*, Band 80 der Reihe *LNI*, Seiten 51–71. GI, 2006.

[WRR07a] Barbara Weber, Stefanie B. Rinderle und Manfred Reichert: *Change Patterns and Change Support Features in Process-Aware Information Systems*. In: John Krogstie, Andreas L. Opdahl und Guttorm Sindre (Herausgeber): *Proceedings 19th International Conference on Advanced Information Systems Engineering (CAiSE 2007), Trondheim, Norway*, Band 4495 der Reihe *Lecture Notes in Computer Science*, Seiten 574–588, London, June 2007. Springer.

[WRR07b] Barbara Weber, Stefanie B. Rinderle und Manfred Reichert: *Identifying and Evaluating Change Patterns and Change Support Features in Process-Aware Information Systems*. Technical Report TR-CTIT-07-22, Enschede, März 2007.

[WRRM08] Barbara Weber, Manfred Reichert und Stefanie Rinderle-Ma: *Change patterns and change support features - Enhancing flexibility in process-aware information systems*. Data Knowl. Eng., 66(3):438–466, 2008.

Lebenslauf

René Wörzberger

Geburtsdatum:	2. Januar 1978
Geburtsort:	Hannover
Geburtsname:	Wörzberger
Staatsangehörigkeit:	deutsch
seit Januar 2005	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik 3 der RWTH Aachen; Beginn der Promotion
Dezember 2004	Studienabschluss als Diplom-Informatiker (Dipl.-Inform.)
1998 – 2004	Informatikstudium an der RWTH Aachen
1997 – 1998	Zivildienst Diakonie/Sozialstation Rösrath-Hoffnungsthal
1997	Abitur am Freiherr-vom-Stein-Gymnasium Rösrath

Aachener Informatik-Berichte

This is the list of all technical reports since 1987. To obtain copies of reports please consult `http://aib.informatik.rwth-aachen.de/` **or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email:** `biblio@informatik.rwth-aachen.de`

1987-01 *	Fachgruppe Informatik: Jahresbericht 1986
1987-02 *	David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 *	Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 *	Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 *	Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 *	Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 *	Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 *	Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 *	Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 *	Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 *	Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12	J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 *	Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik
1988-02 *	Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 *	Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 *	Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 *	Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 *	Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 *	Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 *	Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 *	W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 *	Kai Jakobs: Towards User-Friendly Networking
1988-11 *	Kai Jakobs: The Directory - Evolution of a Standard

1988-12 * Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 * Martine Schümmer: RS-511, a Protocol for the Plant Floor
1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing
1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation
1988-16 * Fachgruppe Informatik: Jahresbericht 1987
1988-17 * Wolfgang Thomas: Automata on Infinite Objects
1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs
1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers
1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen
1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment
1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers
1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies
1988-24 * Uschi Heuter: Generalized Definite Tree Languages
1989-01 * Fachgruppe Informatik: Jahresbericht 1988
1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions
1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language
1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)
1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?
1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems
1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control
1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial
1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation
1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science
1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?
1989-13 * Martine Schümmer: CNC/DNC Communication with MAP
1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements
1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays

1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension

1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages

1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System

1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming

1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages

1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming

1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline

1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes

1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability

1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design

1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems

1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems

1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification

1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs

1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint

1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL

1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems

1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems

1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks

1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases

1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem

1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion

1991-30 T. Margaria: First-Order theories for the verification of complex FSMs

1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications

1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991

1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories
1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes
1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems
1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line
1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme
1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management
1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines
1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking
1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD
1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars
1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)
1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik
1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual
1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems
1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages
1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)
1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine
1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus
1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions
1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code
1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine
1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)

1992-19-08	D. Gärtner, A. Kimms, W. Kluge: pi-Red^+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus
1992-19-09	D. Howe, G. Burn: Experiments with strict STG code
1992-19-10	J. Glauert: Parallel Implementation of Functional Languages Using Small Processes
1992-19-11	M. Joy, T. Axford: A Parallel Graph Reduction Machine
1992-19-12	A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction
1992-19-13	K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)
1992-19-14	K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer
1992-19-15	S. Mintchev: Using Strictness Information in the STG-machine
1992-19-16	D. Rushall: An Attribute Grammar Evaluator in Haskell
1992-19-17	J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation
1992-19-18	W.S. Martins: Parallel Implementations of Functional Languages
1992-19-19	D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)
1992-19-20	J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment
1992-19-21	S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)
1992-19-22	G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers
1992-19-23	S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)
1992-19-24	W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)
1992-19-25	M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting
1992-19-26	Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)
1992-19-27	H. Doerr: Monitoring with Graph-Grammars as formal operational Models
1992-19-28	J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures
1992-19-29	G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)
1992-20	H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language
1992-21	H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language
1992-22	H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing
1992-23	T. Margaria, B. Steffen: Distinguishing Formulas for Free
1992-24	K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 *	R. Stainov: A Dynamic Configuration Facility for Multimedia Communications
1992-26 *	Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification
1992-27	W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety
1992-28 *	Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design
1992-29	B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik
1992-30	A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic
1992-32 *	Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems
1992-33 *	B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications
1992-34	C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice
1992-35	J. Börstler: Feature-Oriented Classification and Reuse in IPSEN
1992-36	M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis
1992-37 *	K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models
1992-38	A. Zuendorf: Implementation of the imperative / rule based language PROGRES
1992-39	P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme
1992-40 *	Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks
1992-41 *	Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems
1992-42 *	P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components
1992-43	W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages
1992-44	N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications
1993-01 *	Fachgruppe Informatik: Jahresbericht 1992
1993-02 *	Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems
1993-03	G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments
1993-05	A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES
1993-06	A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems

1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems

1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars

1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems

1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition

1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents

1994-16 P. Klein: Designing Software with Modula-3

1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages
1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency
1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases
1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views
1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work
1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems
1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming
1996-01 * Jahresbericht 1995
1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees
1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability
1996-05 Klaus Pohl: Requirements Engineering: An Overview
1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs
1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming
1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents
1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines
1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation
1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell
1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems
1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming
1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management
1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement

1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models

1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools

1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases

1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation

1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations

1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs
1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System
1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms
1998-01 * Fachgruppe Informatik: Jahresbericht 1997
1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes
1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr
1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments
1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems
1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future
1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases
1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML
1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
1999-01 * Jahresbericht 1998
1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager
1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference
1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie
1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL
1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-03	D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools
2000-04	Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05	Mareike Schoop: Cooperative Document Management
2000-06	Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 *	Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08	Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java

2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-07 Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19	Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-02	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03	Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04	Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes

2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14	Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15	Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16	George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17	Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18	Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.