6. Arbeitsablaufbasierte Applikationen

Arbeitsablaufpläne, auch Task-Graphen genannt, stellen ein typisches Paradigma zur Spezifikation grobkörniger Applikationen dar, die aus wenigen mächtigen Bausteinen zusammengesetzt sind. Ein solcher Arbeitsablaufplan spezifiziert eine evtl. auch nebenläufige Abarbeitung von Aktionen, die i.a. mit Seiteneffekten behaftet sind. In UML werden sie z.B. in Form von Aktivitätsdiagrammen zur Modellierung von Systemverhalten eingesetzt.

In diesem Kapitel wird ein Erweiterungsmodul vorgestellt, das sich als generische Grundlage für andere Erweiterungsmodule eignet, welche Applikationsentwicklung für ein konkretes System unter Einsatz von Arbeitsablaufplänen unterstützen sollen. Neben der Präsentation dieses Erweiterungsmoduls werden hier auch die technischen Grundlagen und Prinzipien dargestellt, die von ECL bei Erweiterungsmodulen eingesetzt werden.

6.1. Architekturvokabular

Abbildung 6.1 zeigt die Elementtypen des Architekturvokabulars als Klassen in einem UML-Klassendiagramm. Die Piktogramme, die den Elementtypen hinzugefügt sind, werden in dem Editor als grafische Repräsentation der Elemente benutzt. Wie schon erwähnt, werden die Elementtypen in der Architekturbeschreibungssprache Acme [GMW97] definiert, die ein objektorientiertes Typsystem benutzt. Zwei zentrale Grundtypen sind Komponenten und Konnektoren. Architekturen werden erstellt, indem Komponenten über Konnektoren verbunden werden. Sie bilden somit einen bipartiten Graph. Die Verbindungspunkte der Komponenten sind die Anschlüsse (`Port`), die Verbindungspunkte der Konnektoren heißen Rollen (`Role`).

Konnektoren vom Typ `Start` initiieren eine Applikation. Beim Applikationstart senden sie ein Signal an alle verbundenen Komponenten. Um eine einfache Komposition von evtl. rekursiv spezifizierten Architekturelementen zu erlauben, können in einer Applikationen beliebig viele `Start`-Konnektoren existieren. Die Komponenten, die mit einem `Start`-Konnektor verbunden sind und Aktivierungssignale empfangen, befinden sie sich in der Rolle eines Empfängers. Daher trägt die Rolle des `Start`-Konnektors den Namen `rcv`, für Receiver.

Der Typ `End` dient zur Terminierung der gesamten Applikation. Wenn mehrere `End`-Konnektoren in einer Applikation vorhanden sind, terminiert die Ap-

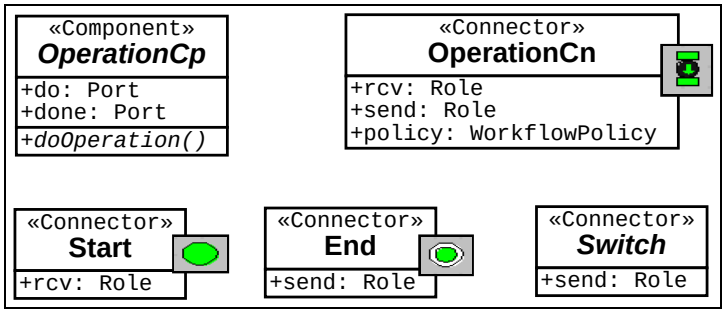

Abbildung 6.1.: Vokabular für arbeitsablaufbasierte Applikationen mit ihren grafischen Repräsentationen im Editor

plikation bei der ersten Aktivierung eines `End`-Konnektors. Analog zu der obigen Argumentation trägt die Rolle des `End`-Konnektors den Namen `send`.

Der abstrakte Komponententyp `OperationCp` stellt eine abstrakte Operation zur Verfügung. Eine Operation ist eine Aktion, die angestoßen wird, eine Berechnung oder Interaktion durchführt und sich selbst beendet. Andere Erweiterungsmodule, die auf diesem Modul aufsetzen, stellen konkrete Operationen durch Spezialisierung von `OperationCp` zur Verfügung.

`OperationCp` besitzt die zwei Anschlüsse `do` und `done`. Um das Verhalten der Komponenten zu modellieren, wird im Folgenden ein Ereignismodell eingesetzt. Komponenten/Konnektoren können an ihren Verbindungspunkten Signale empfangen und gezielt Signale zu bestimmten Verbindungspunkten verbundener Komponenten/Konnektoren senden.

Abbildung 6.2 spezifiziert das Verhalten von `OperationCp` als Zustandsmaschine in UML-Notation. Dabei wird eine lokale Variable *i* zur Pufferspezifikation eingesetzt. Wird ein Signal an dem Anschluss `do` empfangen, wird die konkrete Operation als Eingangsaktion des Zustands `active` nebenläufig durchgeführt. Weiter eingehende `do`-Signale werden gepuffert. Nach der Terminierung der Operation wird ein Signal an einem internen `term`-Anschluss empfangen. Daraufhin wird in der Ausgangsaktion ein Signal an alle Rollen gesendet, die mit `done` verbunden sind. Sind während der Ausführung der Operation über den Anschluss `do` weitere Signale eingegangen und ist damit `i` größer als null, wird die Operation entsprechend oft wiederholt ausgeführt. Dieses Verhalten garantiert, dass kein Aktivierungssignal verloren geht und immer nur maximal eine Operation ausgeführt wird.

Operationskomponenten werden über Konnektoren vom Typ `OperationCn` miteinander verbunden. Dieser besitzt die zwei Rollen `rcv` und `send`. Die Rolle `send` wird mit den `done` Anschlüssen von `OperationCp`-Komponenten verbunden, da sich diese in der Rolle eines Senders von Ereignissen befinden. Dementsprechend werden `rcv`-Rollen mit `do`-Anschlüssen verbunden.

Das Verhalten eines `OperationCn`-Konnektors beim Eintreffen von Signalen hängt von der gewählten Verhaltensstrategie ab, die in dem Feld `policy` spezifiziert ist. Die folgenden zwei Verhaltensstrategien werden unterstützt:

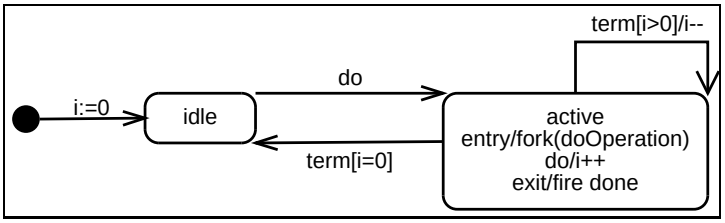

Abbildung 6.2.: Verhaltensspezifikation für Operationskomponenten

- Bei der Politik `Barrier` wird gewartet, bis von jeder Komponente, die mit der Rolle `send` verbunden ist, ein Signal geschickt wurde. Erst dann wird ein Signal an alle mit `rcv` verbundenen Komponenten geschickt. Es wird also eine Barrierensynchronisation durchgeführt.

- Die Politik `Join` schickt bei jedem Eintreffen eines Signals an der `send` Rolle ein Signal an alle mit `rcv` verbundenen Komponenten. Diese Politik wird eingesetzt, wenn alternative Kontrollflüsse wieder zusammengeführt werden.

Prinzipiell sind beliebig viele andere zusätzliche Verhaltensstrategien denkbar, in den bisherigen Applikationen haben sich die beiden obigen aber als ausreichend erwiesen.

Abbildung 6.3 spezifiziert die strukturellen Einschränkungen für die Komposition von `OperationCp` und `OperationCn`. Die ersten zwei Regeln besagen, dass jede `OperationCp`-Komponente nur maximal mit einem `OperationCn`-Konnektor an `rcv` oder `send` verbunden werden kann. Die dritte und vierte Regel besagen, dass ein `do`-Anschluss einer Komponente nur mit einer `rcv`-Rolle eines Konnektors und ein `done`-Anschluss nur mit einer `send`-Rolle verbunden werden kann. Damit ist gewährleistet, dass die Koordination des Kontrollflusses bei den Konnektoren konzentriert ist. Sie müssen auf mehrere einkommende Ereignisse von unterschiedlichen Quellen reagieren und diese an die richtigen Empfänger weiterleiten. Die Komponenten dagegen kapseln hauptsächlich eine konkrete Aktion.

Ein wichtiges Element für Entscheidungspläne sind bedingte Alternativen. Diese steuern den Kontrollfluss anhand von Bedingungen an den Systemzustand, die von dem Applikationsentwickler spezifiziert wurden. Nun hängt der Typ des Systemzustands sehr von den Erweiterungsmodulen ab, die der Entwickler für seine Applikation gewählt hat. Benutzt er z.B. ein Modul für CORBA-basierte Applikationen, dann ist es sinnvoll, den Kontrollfluss anhand von Rückgabewerten eines Methodenaufrufs auf einem entfernten CORBA-Objekt zu steuern. Benutzt er dagegen ein Modul, das ein eigenes Speichersystem mit typisierten Datenobjekten verwaltet, lässt sich der Kontrollfluss anhand der aktuellen Werte eines Datenobjekts steuern.

Daher enthält das Erweiterungsmodul für ablaufbasierte Applikationen nur den abstrakten Konnektortyp `Switch`. Erweiterungsmodule, die einen geeigneten Systemkontext anbieten, können konkrete Spezialisierungen enthalten.

```
style Operations = {
  invariant forall cp in self.components |
     declaresType(cp,OperationCp) ->
        ( // Regel 1
          size({select cn in self.connectors | declaresType(cn,OperationsCn)
                                                and
                                               attached(cn.send,cp.done)})
          <=1

           and
          // Regel 2
          size({select cn in self.connectors | declaresType(cn,OperationsCn)
                                                and
                                               attached(cn.rcv,cp.do)})
          <=1

           and
          // Regel 3
          forall cn in self.connectors |
             forall p in cp.ports | (attached(cn.send,p)-> p == cp.done)
                                     and
                                    (attached(cn.rcv,p)-> p == cp.do)

        );
  invariant forall cn in self.connectors |
     declaresType(cn,OperationCn) ->
        // Regel 4
        forall cp in self.components | (forall p in cp.ports |
                                          (attached(cp.do,p)-> p == cn.rcv)
                                            and
                                          (attached(cp.done,p)-> p == cn.send)
                                        );

};
```

Abbildung 6.3.: Strukturelle Einschränkungen der Komposition von `OperationCp` und `OperationCn` spezifiziert mit Armani-Invarianten

Dieses Erweiterungsmodul bietet damit eine reine Steuerung des Kontrollflusses. Eine Erweiterung um Datenflusselemente ist auf den ersten Blick sehr einfach, da nur bei jeder Aktivierung neben dem eigentlichen Aktivierungssignal weitere Daten hinzugefügt werden müssen. Das Problem besteht nun darin, dass für diese Daten ein Typsystem gefunden werden muss, das sich für alle möglichen Operationen eignet. Da diese aber prinzipiell beliebig sein können, würde jedes Typsystem die Menge möglicher Operationen einschränken. Deswegen wurde auf eine umfassende Unterstützung von Datenflusselementen hier verzichtet.

Da es aber doch hilfreich ist, wenn eine Operation bei ihrer Aktivierung auch weitere Parameter von ihrem Vorgänger erhält, ist es möglich, einem Aktivierungssignal eine Zeichenkette als Inhalt hinzuzufügen. Diese Möglichkeit sollte aber nur für einfache kleine Nachrichten eingesetzt werden. Wenn ein `OperationCn`-Konnektor eine Aktivierung an alle mit ihm verbundenen Komponenten weitergibt, erhält jede Komponente die gleiche Nachricht. Ist der Konnektor mit der `Join` Verhaltensstrategie parametrisiert, wird jede eingehende Nachricht weitergegeben. Besitzt er dagegen die `Barrier` Strategie, wird die Nachricht der zuletzt eingegangenen Aktivierung benutzt. Alle anderen eingegangenen Nachrichten verfallen.

Andere Erweiterungsmodule enthalten fortgeschrittenere Möglichkeiten für den Datenaustausch zwischen Operationen. In Kapitel 8.1 wird ein Erweiterungsmodul vorgestellt, das es Operation ermöglicht, Daten in entfernten Datenobjekten zu speichern und so über diese Datenobjekte zu kommunizieren. In Kapitel 8.2 werden Konnektoren eingeführt, die allgemeine Datenströme miteinander verknüpfen können. In Kapitel 12.2.2 wird eine Spezialisierung des `OperationCn`-Konnektors präsentiert, die es ermöglicht, Daten mit CORBA-Typen in ein Aktivierungssignal einzubinden.

6.2. Laufzeitunterstützung

Die Laufzeitunterstützung für arbeitsablaufbasierte Applikationen basiert auf einem einfachen Kontrollflussgraph. Abbildung 6.4 zeigt die wichtigsten Klassen. Jeder Knoten hat seinen eigenen Kontrollfluss, einen Verweis auf die Laufzeitunterstützung für arbeitsablaufbasierte Applikationen (`Workflow-AppExtension`) und einen Verweis auf die gesamte Applikation (`AppGraph`) mit all ihren Laufzeitunterstützungen.

Bei dem Start einer Applikation werden, wie in Kapiteln 5.2.3 schon erklärt wurde, zuerst alle Laufzeitunterstützungen gestartet. Diese suchen dann nach Aufgaben, die sie bearbeiten können. Die Laufzeitunterstützung dieses Erweiterungsmoduls für arbeitsablaufbasierte Applikationen sucht in den Applikationsobjekten der Applikation zuerst nach allen Objekten der Klasse `SimpleNode` und initialisiert diese. Dabei wird u.a. für jedes Objekt ein eigenes Thread-Objekt erzeugt und gestartet, welches dann die Aktivierungssignale bedient, die an ein `SimpleNode`-Objekt gesendet werden.

Danach sucht die Laufzeitunterstützung nach allen Objekten der Klasse `StartNode` und aktiviert diese. Ein `StartNode`-Objekt hat als Spezialisie-

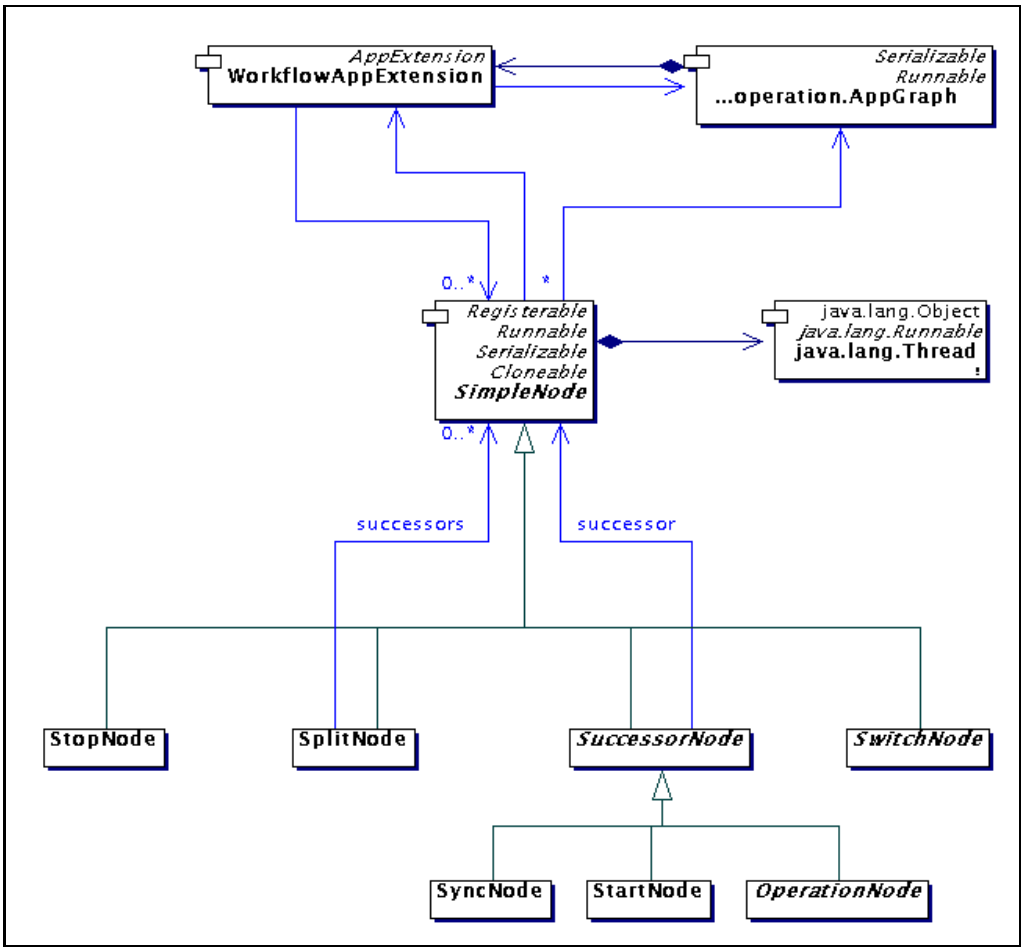

Abbildung 6.4.: Implementierungsklassen der Laufzeitunterstützung für arbeitsablaufbasierte Applikationen

rung von `SuccessorNode` genau einen Nachfolger, an den er bei eigener Aktivierung die Aktivierung weitergibt.

In der Applikationsarchitektur wird ein `Start`-Konnektor i.a. mehrere Nachfolger haben. In diesem Fall erzeugt der Elementübersetzer für `Start`-Konnektoren zwei Applikationsobjekte: ein `StartNode`-Objekt verknüpft mit einem `SplitNode`-Objekt als Nachfolger. Ein solches `SplitNode`-Objekt hat beliebig viele Nachfolger, an die es eine eigene Aktivierung in nichtdeterministischer Reihenfolge weitergibt.

Ein `OperationCn`-Konnektor, dessen `rcv`- und `send`-Rollen mit beliebig vielen `OperationCp`-Komponenten verbunden sein können, wird ebenfalls zu zwei verknüpften Applikationsobjekten übersetzt, einem `SyncNode`- und einem `SplitNode`-Objekt. Ebenso wie bei dem `Start`-Konnektor dient das `SplitNode`-Objekt dazu, die Aktivierung an alle Nachfolger weiterzugeben. Das `SyncNode`-Objekt ist als Nachfolger mit den Applikationsobjekten der Komponenten verbunden, die mit der `send`-Rolle verknüpft sind. Es verwaltet die eingehenden Aktivierungen analog zu der für den `OperationCn`-Konnektor in dem Feld `policy` gewählten Verhaltensstrategie.

Operationen werden von spezialisierten `OperationNode` Knoten durchgeführt. Die Oberklasse stellt das im vorherigen Kapitel beschriebene Verhalten zur Verfügung. Da eine `OperationCp`-Komponente stets nur mit höchstens einem Konnektor für eingehende Aktivierungen und einem Konnektor für ausgehende verbunden sein kann, besteht die Übersetzung aus dem Erzeugen eines geeigneten Applikationsobjekt, welches mit dem entsprechenden `SplitNode`- und `SyncNode`-Objekt verbunden wird.

Die abstrakte Klasse `SwitchNode` enthält nur sehr wenig Funktionalität zusätzlich zu ihrer Oberklasse. Konkrete Unterklassen aus anderen Erweiterungsmodulen müssen diese bereitstellen, um einen bedingten Kontrollfluss zu unterstützen.

Die Laufzeitunterstützung terminiert, wenn ein `StopNode`-Objekt aktiviert wird. Wenn alle Laufzeitunterstützungen terminiert sind, terminiert die Applikation.

Das Laufzeitsystem für ablaufbasierte Applikationen unterstützt damit noch keine Verteilung. Alle Operationen werden in einem Adressraum koordiniert. Die Operationen selber können allerdings entfernte Ressourcen benutzen. In Kapitel 9 werden konkrete Operationen vorgestellt, die entfernte Rechendienste benutzen.