5. Ein erweiterbarer, architekturbasierter Ansatz

In diesem Abschnitt wird ein architekturbasierter Ansatz für einen erweiterbaren Rahmen eines Koordinationssprachensystems namens *Extensible Coordination Language (ECL)* vorgestellt, der sich zur Erstellung verteilter Applikationen für unterschiedliche Zielplattformen eignet, wie z.B. CORBA und Grid-Computing. Verschiedene Zielplattformen und Anwendungsbereiche werden dabei von ECL-Erweiterungsmodulen unterstützt. Applikationsentwickler wählen sich aus einer Bibliothek für ihre Applikationen die benötigten Erweiterungsmodule und können so auch die spezifischen Besonderheiten der jeweiligen Zielplattform nutzen. Heterogene Applikationen, die gleichzeitig unterschiedliche Zielplattformen einsetzen, lassen sich unter Kombinierung der entsprechenden Erweiterungsmodule erstellen. Applikationsentwickler erstellen eine Anwendung als einen Graphen bestehend aus Komponenten und Konnektoren, die von den Erweiterungsmodulen zur Verfügung gestellt werden.

Eine fiktive Anwendung aus dem Bereich der Kontrollsysteme soll diesen Ansatz verdeutlichen. In einem modernen Haus seien die elektronisch steuerbaren Geräte, wie z.B. Heizung, Rollläden, Stereoanlage etc., über ein Netzwerk miteinander verbunden. Zusätzlich sind Sensoren für Helligkeit, Temperatur, Luftfeuchtigkeit etc. installiert und ebenfalls mit dem Netzwerk verbunden. Jedes Gerät ist über eine CORBA-Schnittstelle steuerbar[1].

Jedes System muss speziell vor Ort von einem Beauftragten des Vertreibers eingerichtet werden. Hierfür wird zunächst die Hardware installiert. Danach müssen die einzelnen Geräte konfiguriert werden. Insbesondere müssen die richtigen Sensoren mit den korrekten Aktoren verbunden werden. So könnte jeder Raum einen eigenen Helligkeitssensor besitzen, dessen Messdaten für die Steuerung der Jalousien und Lampen in diesem Raum benutzt werden.

Der Beauftragte setzt für die Konfiguration ein von dem Vertreiber der Geräte erstelltes Erweiterungsmodul ein, das für jeden Gerätetyp einen entsprechenden Komponententyp zur Verfügung stellt. Zusätzlich enthält diese Erweiterung auch Konnektortypen zur Konfiguration. Mit diesen Typen kann der Beauftragte einen Graph erstellen, der die Konfiguration des Gesamtsystems spezifiziert. Aus diesem Graph erzeugt das ECL-System dann

[1] Natürlich ist in einem realen System nicht davon auszugehen, dass jedes einzelne Gerät einen Kleinrechner für die Unterstützung von CORBA enthält. Es wird einen Zentralrechner geben, welcher die Geräte direkt ansteuert. Die Open-Services-Gateway-Initiative verfolgt z.B. genau diesen Ansatz [osg00]. Dieser Zentralrechner kann über CORBA eine entfernte Steuerung der einzelnen Geräte ermöglichen.

ein ausführbares Programm, welches das Gesamtsystem aufsetzt.

Nun ist es denkbar, dass der Besitzer des Hauses zu einem späteren Zeitpunkt sein System um zusätzliche Fähigkeiten erweitern möchte. So wünscht er sich z.B. ein spezielles abendliches Erholungsprogramm, das er bei Bedarf aktivieren kann. Dieses soll die Lampen dimmen, sich durch Zugriff auf das Internet über Webdienste die aktuellen Musikcharts besorgen und diese über die Stereoanlage abspielen. Der Besitzer beauftragt für die Erstellung dieses Programms einen Entwickler einer spezialisierten Softwarefirma.

Der Entwickler kann das von dem Vertreiber mitgelieferte Erweiterungsmodul für die Ansteuerung der einzelnen Geräte einsetzen. Zusätzlich hat er Zugriff auf den Graph, der die Konfiguration des Haussystems beschreibt, und weiß daher, welche Geräte es gibt und wie er sie lokalisieren kann. Das geforderte Erholungsprogramm ist aber keine Konfiguration von Komponenten, sondern besteht aus einem Ablaufplan einzelner Operationen. Daher sind die in dem mitgelieferten Erweiterungsmodul vorhandenen Konnektortypen nicht ausreichend.

Das ECL-Standardsystem hat aber ein allgemeines Erweiterungsmodul, zur Spezifikation solcher Ablaufpläne. Dieses kann der Entwickler benutzen, wobei er es evtl. noch an das Haussystem anpassen muss. Zusätzlich benötigt er ein Erweiterungsmodul für den Zugriff auf Webdienste. Mit diesen Erweiterungsmodulen kann er nun einen Graph erstellen, welcher das gewünschte Programm als Ablaufplan spezifiziert. Das ECL-System erzeugt daraus ein ausführbares Programm, welches an geeigneter Stelle im Haussystem installiert werden muss.

Dieses Szenario ist mit dem ECL-System realisierbar [FO03]. Das nächste Kapitel gibt einen allgemeinen Überblick. Kapitel 5.2 beschreibt die einzelnen Phasen der Applikationsentwicklung. In Kapitel 5.3 werden die unterschiedlichen Arten zur Erweiterung von ECL vorgestellt. Kapitel 5.4 weist Möglichkeiten auf für Interaktion unterschiedlicher Erweiterungsmodule. Abschließend diskutiert Kapitel 5.5 kurz, wie sich Applikationen analysieren lassen. In den Kapiteln 6, 8, 9, 11 und 12 werden konkrete ECL-Erweiterungsmodule ausführlich vorgestellt.

5.1. Überblick über ECL

Abbildung 5.1 gibt einen Überblick über das ECL-System. Zuerst wählt der Applikationsentwickler einen Satz von Erweiterungsmodulen aus, die er für seine Applikation benötigt. Mit diesen erstellt er dann seine Applikation und übersetzt sie. Der Anwender führt sie dann aus.

Ein Erweiterungsmodul kann Elemente anderer Erweiterungsmodule einsetzen, so dass es eine Abhängigkeitsbeziehung zwischen den Modulen gibt. Ein Erweiterungsmodul enthält die folgenden Bestandteile:

- *Architekturvokabular*
  Das Vokabular besteht aus einer Sammlung von Komponenten- und Konnektortypen, die von einem Sprachentwickler in der Architekturbeschreibungssprache Acme [GMW97] definiert sind. Applikationsent-

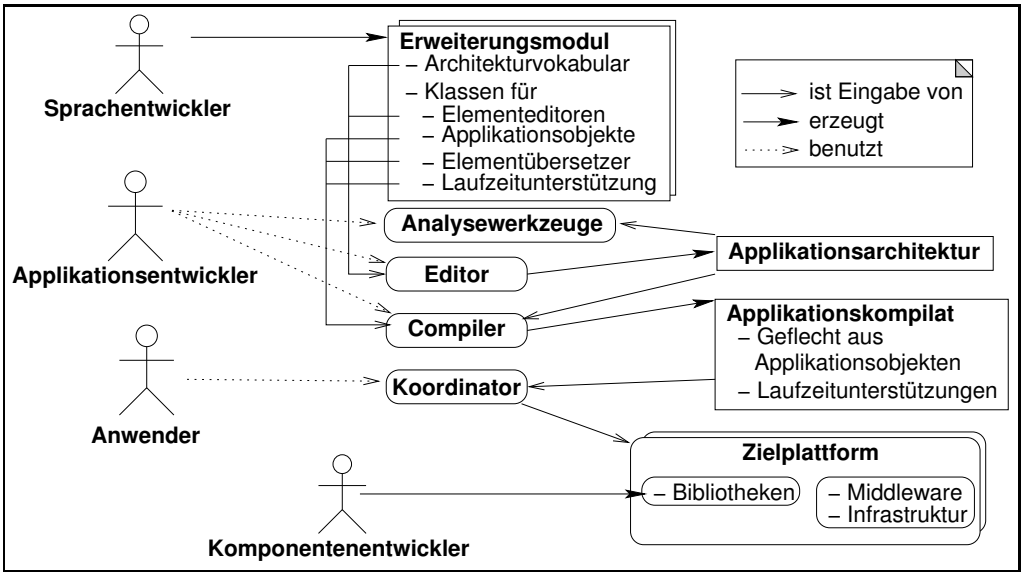

Abbildung 5.1.: Übersicht über das ECL System

wickler erstellen eine Applikation, indem sie für gegebene Typen Architekturelemente erzeugen, diese geeignet parametrisieren und miteinander verbinden. Elementtypen können hierarchisch definiert sein, d.h. sie können selber intern ein System aus Komponenten und Konnektoren enthalten. Ein Beispiel ist ein Komponententyp für Subsysteme, welcher es dem Applikationsentwickler ermöglicht, seine Applikation hierarchisch zu gliedern. Hierfür verpackt er die Subsysteme der Applikation in eine Komponente. Die Applikation besteht dann aus wenigen verbundenen Subsystemkomponenten, die intern dann weitere Komponenten enthalten.

- *Elementeditorklassen*
  Der Editor für Applikationen benutzt für die Bearbeitung der einzelnen Applikationselemente spezielle Elementeditoren. Erweiterungsmodule können dem Editor für die Elementtypen aus ihrem Architekturvokabular spezialisierte Elementeditorklassen anbieten, die in Java implementiert sind. Dadurch lassen sich an die Elementtypen angepasste grafische Schnittstellen in den allgemeinen Editor integrieren. Für einen Komponententyp, der ein lokales Programm startet, kann z.B. eine Schnittstelle realisiert werden, die nur den Namen dieses Programmes abfragt, dem Applikationsentwickler evtl. auch eine textuelle Erklärung dieses Parameters anbietet und schließlich den Namen syntaktisch überprüft. Der allgemeine Komponenteneditor, der benutzt wird, wenn keine spezielle Elementeditorklasse gefunden wird, ermöglicht es bei einer Komponente beliebige Eigenschaften auf beliebige Werte zu setzen. Spezielle Elementeditorklassen können daher den Bedienungskomfort erhöhen und die Wahrscheinlichkeit für Fehler reduzieren, die aus falschen Parame-

tern resultieren.

- *Applikationsobjektklassen*
  Die Klassen für Applikationsobjekte stellen die Semantik der einzelnen Architekturelementtypen dar. Applikationsobjekte werden für die Architekturelemente einer Applikation erzeugt und führen Teile der Applikation entsprechend der Konfiguration der Architekturelemente aus.

- *Elementübersetzerklassen*
  Um komplette Applikationen übersetzen zu können, benötigt das ECL-System Übersetzer für einzelne Architekturelemente. Ein Elementübersetzer übersetzt ein Architekturelement in ein Geflecht von Applikationsobjekten. Elementübersetzer können nur Elemente übersetzen, deren Typ aus dem Architekturvokabular des Erweiterungsmoduls stammt.

- *Laufzeitunterstützungsklassen*
  Die Laufzeitunterstützung koordiniert die Ausführung der Applikationsobjekte einer übersetzten Applikation. Wenn das Erweiterungsmodul für ein bestimmtes Zielsystem entwickelt wurde, bietet die Laufzeitunterstützung Schnittstellen an, die den Applikationsobjekten einen Zugriff auf das Zielsystem ermöglichen.

Die Architekturelementtypen können neben der Funktionalität auch weitere Informationen beinhalten, die von externen Analysewerkzeugen verarbeitet werden können. So ist es beispielsweise möglich, das abstrakte Verhalten einer Komponente über endliche Automaten zu spezifizieren, bzw. Aussagen über ihren Ressourcenverbrauch zu treffen. Hinzukommend können die Module Regeln enthalten, die strukturelle Beschränkungen für Applikationen, welche die angebotenen Architekturelemente einsetzen, spezifizieren.

Es existiert ein grundlegendes Basismodul, das von allen Erweiterungsmodulen benutzt wird. Es enthält abstrakte Basisklassen für Elementeditoren und -übersetzer, welche die Erstellung spezieller Klassen stark vereinfacht. Das Basismodul enthält keine Architekturelemente.

Eine solche Applikationsarchitektur wird unter Einsatz der Elementübersetzer zu einem einfachen Behälter von sogenannten Applikationsobjekten übersetzt. Diese enthalten Code, der unter Einsatz der Laufzeitunterstützungen auf das Zielsystem zugreift. Eine ausführbare Applikation, ein Applikationskompilat, besteht aus diesen Applikationsobjekten und aus allen Laufzeitunterstützungen aller Erweiterungsmodule, die von dem Applikationsentwickler für die Erstellung dieser Applikation gewählt wurden.

Die Ausführung des Applikationskompilats wird von einem Koordinator durchgeführt. Hierfür startet er zuerst alle Laufzeitunterstützungen. Diese führen dann die Applikationsobjekte und damit die eigentliche Applikation aus.

Damit große Applikationsarchitekturen handhabbar sind, müssen sie hierarchisch strukturiert sein. Es stellt sich daher die Frage, ob nicht das Architekturvokabular des Basismoduls eine rekursive Komponente enthalten sollte,

die eine interne Architektur enthält, welche von außen nicht direkt sichtbar ist. Dies ermöglicht sowohl eine hierarchische Gliederung als auch die Wiederverwendung von Architekturen als Subarchitekturen. Die eingesetzte Architekturbeschreibungssprache Acme bietet die Möglichkeit mit internen Repräsentationen eine solche Hierarchisierung zu verwirklichen. Daher lässt sich fordern, dass das Basismodul einen allgemeinen rekursiven Komponenten- und Konnektortyp enthält, der sich in allen Erweiterungsmodulen einsetzen lässt. Weitere Überlegungen, die von Erfahrungen aus den Fallstudien gestützt werden, zeigen aber, dass ein solch allgemeiner Typ nur einen geringen praktischen Nutzen bietet.

Das Problem besteht darin, dass jedes rekursive Element konkret eingesetzt werden muss. Angenommen, man möchte ein System aus Komponenten und Konnektoren zu einer fernaufrufbaren Komponente zusammenfassen. Dann muss diese in dem grafischen Editor auch als solche erkennbar sein. Als zweites Beispiel sei eine Architektur, die einen Ablaufplan beschreibt, zu einem Architekturelement zusammengefasst, das wie eine einfache Operation benutzt werden kann. Dieses rekursive Element ist damit eine Mischung aus einem allgemeinen rekursiven Element und einem Operationselement. Eine allgemeine Darstellung als rekursive Komponente wäre unzureichend, da sich die beiden Elemente visuell voneinander unterscheiden sollten.

Der Elementübersetzer könnte in den beiden obigen Fällen gleich sein, da prinzipiell nur das eingepackte System wieder entfaltet werden muss. Das rekursive Architekturelement wird also durch dessen internes System ersetzt. Nun ist es aber möglich, dass ein rekursives Element mehrere interne Systeme besitzt, von denen eines zur Laufzeit nach unterschiedlichen Strategien gewählt wird. Hierfür wird ein spezieller Elementübersetzer benötigt.

Es ist sogar möglich, dass die Semantik, die mit dem internen System verknüpft ist, eine andere als die obige Substitutionssemantik ist. So ist eine Komponente für Parallelverarbeitung nach dem Farm-Prinzip denkbar. Das interne System beschreibt die Funktion, die nebenläufig auf alle Elemente eines Datenobjekts angewendet wird. Der rekursive Elementtyp beschreibt damit eine dynamische Architektur, da seine Exemplare zur Laufzeit durch soviele Exemplare der internen Architektur ersetzt wird, wie das Datenobjekt Elemente hat. Damit kann ebenso wie bei der grafischen Darstellung der Elementübersetzer für jeden rekursiven Elementtyp unterschiedlich sein.

Deswegen besitzt das Basismodul keinen allgemeinen rekursiven Elementtyp. Allerdings enthält es Funktionalität, welche die Erstellung von Editoren und Übersetzer für solche rekursiven Typen vereinfacht. Beispiele für derartige Elemente sind in den Kapiteln 9.1 und 9.4.2 gegeben.

5.2. Entwicklung von ECL-Applikationen

Wie bei den meisten Entwicklungssystemen besteht die Applikationsentwicklung in ECL aus drei Phasen:

1. Applikationserstellung evtl. unter Einsatz von Analysewerkzeugen,

2. Erzeugung einer ausführbaren Datei durch Übersetzung,

3. Ausführen dieser Datei.

Die folgenden Kapitel beschreiben, wie diese Phasen in ECL umgesetzt sind.

5.2.1. Applikationserzeugung und -bearbeitung

Um eine neue Applikation zu erstellen, muss ein Entwickler neben dem Namen der Applikation die Erweiterungsmodule auswählen, die er einsetzen möchte. Abbildung 5.2 zeigt den Dialog für das Erzeugen einer neuen Applikation. Oben sieht man die Liste aller zur Verfügung stehenden Erweiterungen, während unten weitere Informationen zu dem angewählten Erweiterungsmodul dargestellt werden. Der Applikationsentwickler wählt die gewünschten Module aus, indem er sie markiert. In dem Beispiel ist nur das Modul `ACE` ausgewählt.

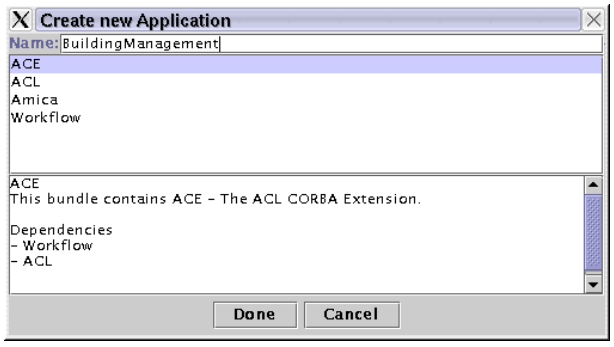

Abbildung 5.2.: Bildschirmfoto des Dialogs zur Erstellung einer neuen Applikation

Wenn der Applikationsentwickler seine Auswahl beendet hat, werden automatisch alle Module, die von den gewählten Modulen benötigt werden, hinzugefügt. Alle Architekturelementtypen, die von den Modulen angeboten werden, werden gesammelt und es wird eine leere Applikationsarchitektur erzeugt. Abbildung 5.3 zeigt einen vereinfachten Ausschnitt dieses Acme-Systems für das Beispiel aus Abbildung 5.2.

Jedes Erweiterungsmodul fasst sein Architekturvokabular, also seine Sammlung von Komponenten- und Konnektortypen, in einem sogenannten Stil (*Style*) zusammen. Das Basismodul ECL enthält nur ein leeres Vokabular. Das Erweiterungsmodul Workflow enthält Typen, um Applikationen in Form von Ablaufplänen zu entwerfen. In der Abbildung ist der Komponententyp `OperationCp` für die Ausführung von Operationen und der Konnektortyp `OperationCn` für die Verknüpfung von Operationen dargestellt. Der Typmechanismus unterstützt auch Mehrfachvererbung, so dass neue Elementtypen auf Elementtypen aus anderen Stilen aufsetzen können. Dies ist an dem Komponententyp `CORBACallCp` aus dem `CORBA`-Stil zu erkennen, der eine Spezialisierung des Komponententyps `OperationCp` ist.

```
Style ECL = { };
Style Workflow extends ECL with
{

  Component Type OperationCp = {
    Port do = new Port;
    Port done = new Port;
    ...
  };

  Connector Type OperationCn = {
    Property policy:string;
    Role rcv = new Role;
    Role send = new Role;
    ...
  };
   ...
  invariant forall cp in
   {select op in self.components |
     declaresType(op,OperationCp}
   | forall cn in
     {select ocn in self.connectors |
       declaresType(op,OperationCn}
     | ( attached(cp,cn) -> ( attached(cn.rcv, cp.do)
                                xor
                              attached(cn.send, cp.done)
       )                    )
};
Style CORBA extends Workflow with
{
  Component Type CORBACallCp extends OperationCp with {
    Property Method : string;
    ...
  };
  ...
};
System BuildingManagement : CORBA =  {};
```

Abbildung 5.3.: Vereinfachter Ausschnitt einer neu erzeugten Applikationsarchitektur in Acme/Armani

Das Verhalten von `OperationCn`-Konnektoren, die mit mehreren Komponenten verbunden sind, wird über das Attribut `policy` definiert. Jedes Architekturelement kann beliebig viele Attribute enthalten. `OperationCp`-Komponenten besitzen zwei Anschlüsse (*Port*) mit den Namen `do` und `done`. Anschlüsse stellen die Schnittstelle einer Komponente dar.

Auch für Anschlüsse ist es möglich, komplexere Typen zu definieren. In diesem Fall wurde aber der Basistyp `Port` gewählt. Über den Anschluss `do` wird die Operation aktiviert, während der Anschluss `done` nach dem Ablauf der Operation benutzt wird, um die nächste Operation zu aktivieren.

Die Schnittstellen von Konnektoren sind Rollen (*Role*). Die Verknüpfung von Komponenten und Konnektoren geschieht immer dadurch, dass ein Anschluss der Komponente mit einer Rolle des Konnektors verbunden wird. Andere Verknüpfungen sind nicht möglich. `OperationCn`-Konnektoren besitzen zwei Rollen `rcv` und `send`, die mit den Anschlüssen der `OperationCp`-Komponenten verbunden werden können.

Die Architekturbeschreibungssprache ermöglicht prinzipiell, dass jeder Anschluss mit jeder Rolle verbunden werden kann. Da dies im allgemeinen nicht sinnvoll ist, kann ein Stil um Invarianten in Form logischer Regeln erweitert werden, welche die zulässigen Verknüpfungen spezifizieren. In dem Beispiel aus der Abbildung ist exemplarisch eine Invariante dargestellt, die besagt, dass, wenn eine `OperationCp`-Komponente mit einem `OperationCn`-Konnektor verknüpft ist, entweder der Anschluss `do` mit der Rolle `rcv` oder der Anschluss `done` mit der Rolle `send` verknüpft ist.

Der Typ `OperationCp` ist abstrakt in dem Sinne, dass keine Semantik mit ihm verknüpft ist. Das Workflow-Erweiterungsmodul enthält also keine Applikationsobjektklassen oder Elementübersetzerklassen, die nicht abstrakt sind. ACE ist ein Erweiterungsmodul, das auf dem Workflow-Modul aufsetzt und den Komponententyp `CORBACallCp` als konkrete Spezialisierung von `OperationCp` enthält. Dieser führt als Operation einen entfernten Methodenaufruf auf einem CORBA-Objekt aus. Der Name der Methode muss von dem Applikationsentwickler als Eigenschaft der Komponente gesetzt werden.

Die Applikationsarchitektur wird in Acme als System dargestellt. Dieses ist nach der Initialisierung noch leer. Für diese leere Applikationsarchitektur mit den zur Verfügung stehenden Elementtypen wird unter Einsatz einer Acme-Bibliothek ein internes Modell erzeugt. Dabei wird die textuelle Repräsentation zu einem Objektgeflecht übersetzt, das Zugriff auf die einzelnen Elemente und Typen erlaubt.

Für das interne Modell wird eine grafische Schnittstelle für den Applikationsentwickler gestartet. Abbildung 5.4 zeigt ein Bildschirmfoto mit dieser Schnittstelle für eine Applikation, die schon einige Architekturelemente enthält. Die dem Entwickler zur Verfügung stehenden Elementtypen eines Erweiterungsmoduls werden in entsprechenden Pull-Down-Menüs bereitgestellt. Wird ein Elementtyp ausgewählt, wird ein neues Element dieses Typs erzeugt und der Applikation hinzugefügt. Der Entwickler kann dieses dann über dessen Verbindungspunkte interaktiv mit den anderen Elementen verbinden.

Mit einem Doppelklick auf ein Architekturelement erhält man einen Editor-

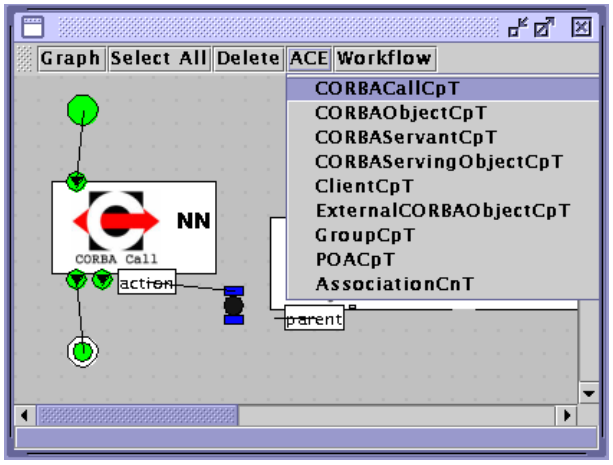

Abbildung 5.4.: Bildschirmfoto der grafischen Schnittstelle zur Konstruktion einer Applikation

dialog, welcher die Parametrisierung des Elements ermöglicht. Dieser Dialog wird anhand des Elementtyps erzeugt und erlaubt so eine typspezifische Manipulation.

5.2.2. Übersetzung

Nachdem der Applikationsentwickler in der ersten Phase die Applikationsarchitektur entworfen hat, wird diese nun zu einem Applikationskompilat übersetzt. Dabei werden die einzelnen Architekturelemente auf Applikationsobjekte abgebildet, welche zusammen mit einer Menge von Laufzeitunterstützungen das Applikationskompilat darstellen.

Der Übersetzungsalgorithmus für Applikationsarchitekturen basiert auf Elementübersetzern, die für einzelne Applikationselemente Applikationsobjekte erzeugen und miteinander vernetzen können. Die Klassen für Elementübersetzer und Applikationsobjekte werden von den Erweiterungsmodulen bereit gestellt.

Die einzelnen Übersetzungsphasen sollen an einem Beispiel erklärt werden. Abbildung 5.5 enthält oben links einen Ausschnitt einer Applikationsarchitektur. Dieser besteht aus einem Ablaufplan, in dem vier Operationen miteinander verbunden sind. Operation 1 und 2 werden nebenläufig ausgeführt. Wenn beide beendet sind, werden gleichzeitig die Operationen 3 und 4 gestartet. Die Übersetzung dieses Ausschnitts der Applikation vollzieht sich nun in den folgenden vier Phasen:

1. Erzeuge für jedes Architekturelement e einen Elementübersetzer C_e. Diese Übersetzer werden in einer Abbildung *compilers* gespeichert, welche Elemente auf ihre Übersetzer abbildet. Jeder Elementübersetzer C_e kann über die Referenz auf das Architekturelement direkt auf die gesamte Applikationsarchitektur und damit auf alle anderen Architekturelemente

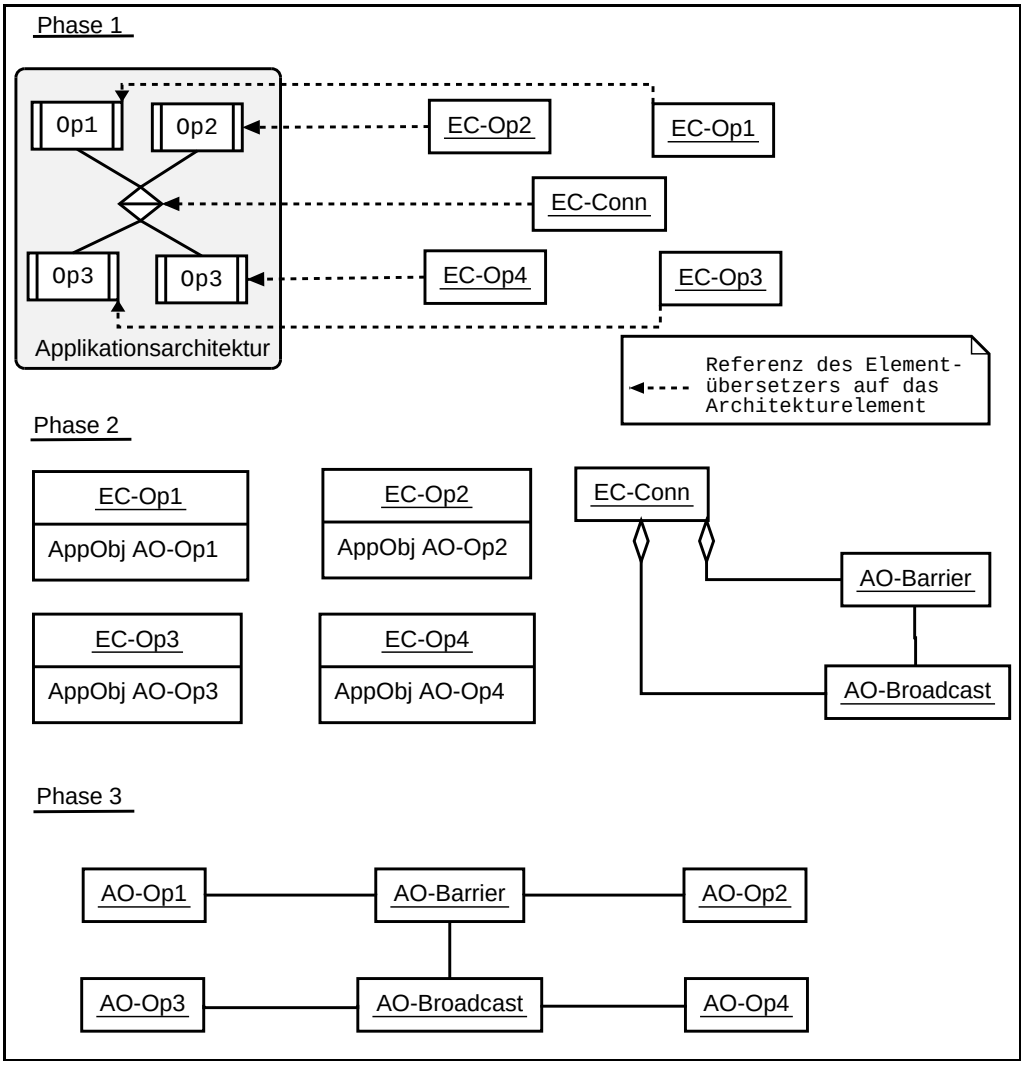

Abbildung 5.5.: Schematische Darstellung der ersten drei Übersetzungsphasen an einem Beispiel

zugreifen. Über die Abbildung *compilers* kann C_e damit Referenzen auf alle anderen Elementübersetzer C_x erhalten, deren Elemente x mit dem eigenen Element e verbunden sind.

2. Erzeuge für jedes einzelne Element die benötigten Applikationsobjekte. Die konkreten Aktionen, die von einem Elementübersetzer ausgeführt werden, hängen von dem ihm zugeordneten Elementtyp ab. In dem Beispiel der Abbildung, werden die Operationen durch ein Applikationsobjekt durchgeführt. Für die Verbindung der Operationen werden dagegen zwei Applikationsobjekte eingesetzt. Eines dient zur Barrierensynchronisation nach der Ausführung von `Op1` und `Op2`, während das andere die beiden Operationen `Op3` und `Op4` nebenläufig aktivieren wird.
   Diese erzeugten Applikationsobjekte werden von jedem Elementübersetzer lokal verwaltet. Sie sind höchstens mit Applikationsobjekten des gleichen Elementübersetzers verbunden, aber nicht mit Applikationsobjekten von anderen Elementübersetzern. So sind die beiden Applikationsobjekte für das Verbinden der Operationen zwar miteinander schon verknüpft aber noch nicht mit den Applikationsobjekten für die Operationen selber.

3. Verbinde alle erzeugten Applikationsobjekte aller Elementübersetzer. Hierfür besitzen die Elementübersetzer eine Methode *wire*, welcher *compilers* übergeben wird, so dass der Zugriff auf die anderen Elementübersetzer und den von diesen erzeugten Applikationsobjekten möglich ist.
   In dem Beispiel werden nun alle Applikationsobjekte miteinander verbunden.

4. Abschließend erzeuge ein Applikationskompilat aus allen erzeugten Applikationsobjekten und den notwendigen Laufzeitunterstützungen. Diese Phase ist in der Abbildung nicht dargestellt.

Für die Erzeugung eines Elementübersetzers werden alle Erweiterungsmodule befragt, ob sie einen Elementübersetzer für den Typ des entsprechenden Architekturelements erzeugen können. Das erste Modul, das dazu in der Lage ist, wird eingesetzt.

Falls kein Elementübersetzer erzeugt werden kann, wird das Element ignoriert. Dieses ist kein selten auftretender Fall, da gerade Konnektoren häufig nur zur Assoziierung von Komponenten eingesetzt werden. In den für die Komponenten generierten Applikationsobjekten werden solche Konnektoren als interne Referenzen realisiert. Für die Konnektoren selber ist daher kein Applikationsobjekt notwendig.

Da ein Elementübersetzer nur Elementtypen des eigenen Erweiterungsmoduls übersetzen kann, kann es nicht mehr als einen Elementübersetzer pro Elementtyp geben. Falls es für einen Typ keinen Übersetzer gibt, wird versucht, einen Übersetzer für einen seiner Supertypen zu finden. Ein Problem besteht nun darin, die richtige Reihenfolge für die Abfrage der Erweiterungsmodule zu finden.

Angenommen ein Erweiterungsmodul B setzt auf einem Modul A auf und bietet einen Architekturelementtyp t_B an, der eine Spezialisierung eines Typs

t_A ist. Ein prinzipiell gültiger Elementübersetzer für ein Architekturelement e vom Typ t_B kann nun sowohl von A als auch von B erzeugt werden. Benötigt wird aber der Elementübersetzer von B, welcher die durch die Spezialisierung ausgedrückte erweiterte Funktionalität unterstützt.

Dieser wird gefunden, indem die Vererbungshierarchie eines Elementtyps rückwärts durchlaufen wird, bis ein Elementtyp gefunden wird, für den ein Elementübersetzer erzeugt werden kann. Da das Typsystem Mehrfachvererbung unterstützt, kann hierbei ein Problem auftreten. Angenommen für einen Typ t_A sei kein Elementübersetzer vorhanden. t_A ist aber eine Spezialisierung von t_B und t_C und für diese beiden Typen existieren Elementübersetzer. Welcher dieser beiden soll benutzt werden? Es gibt zwei Möglichkeiten: beide oder keiner.

Benutzt man beide, werden bei der Übersetzung von Elementen des Typs t_A Applikationsobjekte sowohl für seine von t_B vererbten Attribute erzeugt als auch für die von t_C. Damit würde die Semantik von t_A aus einer Addition der Semantiken von t_B und t_C bestehen.

Eine Addition lässt sich auch immer dadurch erzeugen, dass man ein Element des Typs t_B und ein Element des Typs t_C erzeugt und beide in der Applikationsarchitektur verwendet. Dadurch wird das Verhalten klarer, als wenn ein neues Spezialelement eingesetzt wird, welches evtl. verschiedene Paradigmen kombiniert.

Eine automatische Verknüpfung der Semantiken birgt zusätzlich die Gefahr, dass das Ergebnis unerwartete Eigenschaften hat. Kombiniert man z.B. einen Synchronisationskonnektor für Operationen in einem Ablaufplan mit einem Rundsendungskonnektor, erhält man einen Konnektor, der gleichzeitig synchronisiert als auch Aktivierungssignale sofort weitergibt.

Daher wird in ECL gefordert, dass der Spracherweiterer bei Elementtypen, die sich durch eine kritische Mehrfachvererbung ergeben, explizit eine Semantik durch einen speziellen Elementübersetzer anzugeben hat. Sei $\triangleright$ die Spezialisierungsrelation auf Architekturelementtypen. Es gilt $t_B \triangleright t_A$ genau dann, wenn t_B eine Spezialisierung von t_A ist. Sei γ eine boolsche Funktion auf Elementtypen, die genau dann wahr ist, wenn ein Elementübersetzer für den Elementtyp vorhanden ist. Dann muss nach der obigen Forderung für das benutzte Vokabular folgendes gelten:

$$(\forall t)(\ ((\exists t_1, t_2)(\gamma(t_1) \wedge \gamma(t_2) \wedge t \triangleright t_1 \wedge t \triangleright t_2)) \Rightarrow \gamma(t))$$

Ein weiteres Problem bei der Bestimmung des Elementübersetzers für einen gegebenen Elementtyp besteht darin, dass in dem internen Modell der Applikationsarchitektur, welches mit einer Bibliothek für Acme erzeugt und verwaltet wird, ein Elementtyp keine Referenz auf das Erweiterungsmodul besitzt, dem es entstammt. Das rückwärtige Durchlaufen der Vererbungshierarchie wird dadurch erschwert. Der folgende Algorithmus bestimmt eine korrekte Befragungsreihenfolge der Erweiterungsmodule zur Bestimmung eines Elementübersetzers. Die Verantwortung für die Wahl des korrekten Elementübersetzers, wenn innerhalb eines Moduls mehrere Alternativen vorhanden sind, liegt bei dem Modul selber, dem die Vererbungshierarchie bekannt ist.

Sei M die Menge der Erweiterungsmodule, die von der Applikation benutzt werden, und $used : M \to 2^M$ eine Funktion, die für ein Modul Y die Menge an Modulen $\{X_1 \dots X_n\}$ angibt, die Y direkt benutzt. Mit Benutzung ist hier stets die Spezialisierung von Architekturtypen oder von Klassen für Elementübersetzer etc. gemeint. Die Module X_i können ihrerseits andere Module benutzen. Potenziell kann damit ein Modul X_i Supertypen der Elementtypen von Y besitzen. Da der durch die *used* Funktion bestimmte Abhängigkeitsgraph zyklenfrei sein muss, kann Y keinen Supertyp von einem Elementtyp aus einem X_i enthalten.

Zur Bestimmung der Abfragereihenfolge, wird zuerst die umgekehrte Abhängigkeit $is\_used : M \to 2^M$ berechnet, die für ein Modul X die Menge der Module berechnet, die X benutzen:

$$is\_used(X) = \{Y \in M \mid X \in used(Y)\}$$

Nun läßt sich eine korrekte Reihenfolge, also eine Reihenfolge, welche die obige Problematik stets vermeidet, wie folgt bestimmen:

1. Initialisiere die Ergebnisliste $result := \emptyset$. Initialisiere die Menge *current* mit allen Blättern des durch *used* aufgespannten Baums.

   $$current := \{X \in M \mid is\_used(X) = \emptyset\}$$

2. Füge alle Elemente von *current* in beliebiger Reihenfolge an die Liste *result* an.

   $$result := result + current$$

3. Erstelle eine neue *current* Menge, die aus all den Modulen besteht, die von den Elementen der aktuellen benutzt werden und noch nicht in der Ergebnisliste sind.

   $$current := \{X \in (M \cap result) \mid (\exists Y \in current)(X \in used(Y))\}$$

4. Die Menge *current* kann noch Module enthalten, die andere Elemente aus *current* benutzen. Diese müssen entfernt werden.

   $$current := current / \{X \in current \mid (\exists Y \in current)(Y \in used(X))\}$$

5. Wenn *current* leer ist, terminiere. Anderenfalls springe zurück zu Schritt 2.

Unter der gegebenen Voraussetzung, dass der Abhängigkeitsgraph zyklenfrei ist, terminiert dieser Algorithmus immer. Die Menge *current* beschreibt dann die Wellenfront, die sich ausgehend von den Blättern des Graphs bis zu allen Anfangsknoten ausbreitet.

Mit der in *result* definierten Reihenfolge von dem ersten zu dem letzten Element kann man nun für jedes Element einer Applikationsarchitektur den korrekten Elementübersetzer erzeugen. Die Elementübersetzer für die unterschiedlichen Elementtypen werden in den Kapiteln beschrieben, welche die

Erweiterungsmodule vorstellen. Für rekursive Elementtypen existiert in dem Basismodul allerdings eine Hilfsfunktion, die es ermöglicht, eine Komponente durch ihren Untergraph zu ersetzen. Abbildung 5.6 stellt diese Ersetzung bei der Übersetzung rekursiver Elemente an einem Beispiel dar.

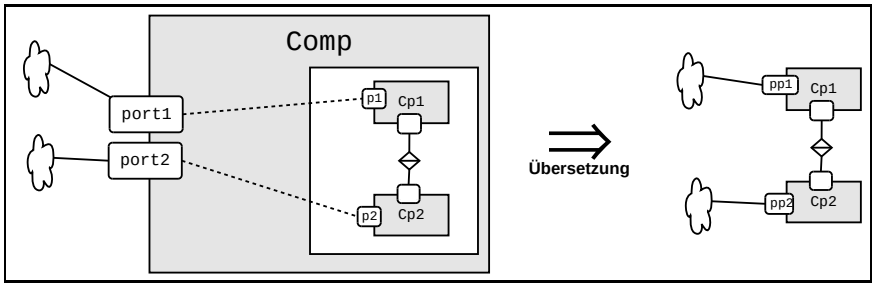

Abbildung 5.6.: Schema der Übersetzung einer rekursiven Komponente

Die Komponente `Comp` enthält eine interne Architektur aus zwei Komponenten. Die interne wird mit der externen Architektur verbunden, indem die externen Anschlüssen `port1`, `port2` explizit den internen Anschlüssen `p1`, `p2` zugeordnet werden. Bei der Übersetzung wird von dem für `Comp` zuständigen Elementübersetzer in der ersten Übersetzungsphase für alle inneren Elemente entsprechende Elementübersetzer erzeugt. In der zweiten Phase wird `Comp` durch die interne Architektur ersetzt. Alle Verbindungen der äußeren Anschlüsse werden dabei auf die assoziierten inneren Anschlüsse umgesetzt. Nun können in der dritten Phase die erzeugten Applikationsobjekte korrekt miteinander verknüpft werden.

Auch wenn dieses Schema einfach ist, so existieren dennoch ein paar Problembereiche. Verbindungspunkte, also Anschlüsse und Rollen, sind selber wieder typisierte Architekturelemente, die Eigenschaften besitzen. Bzgl. des Typs ist bei der Übersetzung auf die Kompatibilität der miteinander assoziierten Verbindungspunkte zu achten. Da der innere Verbindungspunkt an die Stelle des äußeren tritt, muss er den gleichen oder einen spezialisierteren Typ besitzen.

Die Eigenschaften beider Verbindungspunkte werden durch Vereinigung zusammengefasst. Kommt es dabei zu Konflikten, indem beide Punkte eine gleichnamige Eigenschaft mit unterschiedlichen Werten besitzen, so hat der äußere Verbindungspunkt Vorrang. Dies ist dadurch begründet, dass derartige Konflikte am ehesten dadurch entstehen, dass der Applikationsentwickler das eingesetzte Element durch die Eigenschaften der äußeren Verbindungspunkte parametrisiert. Diese von dem Entwickler gewünschte Parametrisierung sollte sich also bei der Substitution durchsetzen[2]. Die Verbindungspunkte nach der Substitution einer Komponente oder eines Konnektors können sich also von den vorherigen Verbindungspunkten unterscheiden. Sie sind daher in der

[2] Allerdings kann es dem Anwendungsentwickler auch durchaus gestattet sein, direkt auf die interne Repräsentation zuzugreifen. So kann er selbst unbeabsichtigt Konflikte erzeugen, indem er die Verbindungspunkte widersprüchlich parametrisiert. Für diesen Fall kann der Elementübersetzer eine Warnung oder einen Fehler ausgeben.

```
1  funct start(args, app) ≡
3     rtcmp := (λ(rt1, rt2) → (rt1.priority ≤ rt2.priority));
4     rts := new SortedSet < RuntimeSupport > (rtcmp);
5     rts := rts ∪ app.runtimeSupports;
6     it := rts.iterator();
7     while (it.hasNext()) do
8           it.next().start(args, app.aos);
9     od
11    wait_until(∀rt ∈ rts)(rt.no_task_left());
13    rts.resort((λ(rt1, rt2) → ¬rtcmp(rt1, rt2)));
14    it := rts.iterator();
15    while (it.hasNext()) do
16          it.next().shutdown();
17    od
18 end
```

Abbildung 5.7.: Algorithmus zur Ausführung einer Applikation vom Koordinator

Abbildung 5.6 mit `pp1` und `pp2` neu benannt worden.

5.2.3. Ausführung

Nachdem ein Applikationskompilat von dem Koordinator geladen wurde, wird es unter Benutzung der enthaltenen Laufzeitunterstützungen ausgeführt. Abbildung 5.7 stellt diesen Algorithmus in freier Pseudocodenotation dar. Neben dem Applikationskompilat selber mit dessen Applikationsobjekten und Laufzeitunterstützungen erhält der Koordinator weitere Argumente, die i.a. von der Kommandozeile stammen.

Zuerst wird jede in der Applikation enthaltene Laufzeitunterstützung, also alle Elemente von `app.runtimeSupports` gestartet, wobei die Parameter von der Kommandozeile und die Applikationsobjekte der Applikation übergeben werden. Hierfür besitzt jede Laufzeitunterstützung die Eigenschaft `priority`, welche die Startreihenfolge bestimmt. In dem Algorithmus ist `rtcmp` eine Vergleichsfunktion auf Laufzeitunterstützungen, die von der sortierten Menge `rts` benutzt wird. In diese werden alle Laufzeitunterstützungen der Applikation eingefügt und dann über den Iterator `it` in der richtigen Reihenfolge gestartet. Dabei werden neben den allgemeinen vom Benutzer übergebenen Parametern `args` einer Laufzeitunterstützung auch mit `app.aos` alle Applikationsobjekte der Applikation übergeben.

Der Einsatz von Prioritäten bedingt, dass der Sprachentwickler sich bei der Erstellung des Erweiterungsmoduls eine geeignete Priorität überlegen muss unter Berücksichtigung der anderen Erweiterungsmodule, mit denen sein Erweiterungsmodul interagiert.

Nachdem alle Laufzeitunterstützungen gestartet wurden, wartet der Koordinator, bis sie alle ihre Aufgaben erledigt haben. Dies ist dann der Fall, wenn bei jeder Laufzeitunterstützung die Methode `no_task_left()` den Wert

`true` liefert. Danach terminiert der Koordinator alle Laufzeitunterstützungen in der umgekehrten Startreihenfolge.

Da sich die Interaktion zwischen Erweiterungsmodulen, die für die Bestimmung der Startreihenfolge relevant ist, nur durch Vererbungsbeziehungen ergibt, wäre auch der Ansatz naheliegend, zuerst die Laufzeitunterstützungen der grundlegenden Erweiterungsmodule und dann nacheinander die Laufzeitunterstützungen der auf diesen aufbauenden Erweiterungsmodule zu starten. Der Ansatz würde mit der Initialisierung von Objekten in einer objektorientierten Sprache korrespondieren, in der zuerst die Konstruktoren der Superklassen ausgeführt werden. Diese Reihenfolge ließe sich dann automatisch bestimmen.

Allerdings eignet sich der Ansatz nicht für abstrakte Erweiterungsmodule, die für die Ausführung die Unterstützung spezialisierter Erweiterungsmodule benötigen. Ein Beispiel für einen derartigen Problemfall ist ein Erweiterungsmodul für Ablaufpläne. Dieses enthält bis auf die eigentlichen Operationen die gesamte notwendige Funktionalität für die Ausführung von Ablaufplänen. Das bedeutet, dass vor der Aktivierung der Laufzeitunterstützung des Ablaufmoduls die Laufzeitunterstützung aller Erweiterungsmodule gestartet werden müssen, die Operationen bereitstellen.

Umgekehrt kann die Laufzeitunterstützung eines spezialisierten Erweiterungsmoduls auch darauf angewiesen sein, dass die Laufzeitunterstützung eines Erweiterungsmoduls, auf das es aufsetzt, schon gestartet ist. So könnte es z.B. Elementtypen enthalten, die spezielle CORBA-Dienste bereitstellen. Um diese zur Laufzeit zur Verfügung stellen zu können, benötigt es die grundlegenden CORBA-Dienste, die von dem Basismodul für CORBA bereitgestellt werden.

Daher muss der Sprachentwickler explizit eine Reihenfolge festlegen. Da ein Erweiterungsmodul i.a. mit wenigen anderen Erweiterungsmodulen interagiert, stellt dies auch keinen zu großen Aufwand dar. Laufzeitunterstützungen erhalten beim Starten als Parameter die dem Koordinator übergebenen Argumente und alle Applikationsobjekte des Applikationskompilats. Eine Laufzeitunterstützung wird in dieser Startfunktion im allgemeinen die Applikationsobjekte nach Aufgaben durchsuchen, die sie zu erledigen hat.

Angenommen eine Laufzeitunterstützung entstammt einem Erweiterungsmodul für Ablaufpläne. Dann wird sie in der Startfunktion nach Applikationsobjekten suchen, die einen Ablaufplan aktivieren. Alle gefundenen Applikationsobjekte werden aktiviert. Diese aktivieren dann wiederum die mit ihnen verbundenen Applikationsobjekte und führen so den Ablaufplan aus. Ob die Startfunktion erst terminiert, wenn alle Ablaufpläne bearbeitet sind, hängt von dem Entwickler des Erweiterungsmoduls ab. Im allgemeinen wird die Startfunktionen früher terminieren, so dass verschiedene Laufzeitunterstützungen nebenläufig ihre Aufgaben bearbeiten können.

5.3. Erweiterbarkeit

Es gibt für Sprachentwickler mehrere Möglichkeiten neue Erweiterungsmodule zu implementieren. Welche gewählt wird, hängt davon ab, auf welche schon vorhandene Funktionalität der Sprachentwickler aufbauen kann. Das ECL-System ist derart konzipiert worden, dass möglichst viele Elemente der Erweiterungsmodule bei der Erzeugung neuer Erweiterungsmodule wiederverwendet werden können. Um den Implementierungsaufwand für die konkrete Realisierung eines Erweiterungsmoduls abschätzen zu können, wird auf das Kapitel 5.6 verwiesen, in dem die Implementierung von ECL beschrieben wird. Im folgenden werden nur die unterschiedlichen Konzepte vorgestellt, um neue Architekturelementtypen mit den zugehörigen Elementeditoren, Elementübersetzern und Applikationsobjekten zu entwickeln.

Komposition vorhandener Architekturelemente

Ein Standardansatz bei der Programmentwicklung ist die funktionale Dekomposition, die daraus besteht, dass ein komplexes Problem in weniger komplexe Teilprobleme zerlegt wird. Diese werden wiederum zerlegt, bis die Teilprobleme so einfach geworden sind, dass sie sich einfach lösen lassen. Diese Teillösungen werden dann so zusammengesetzt, dass sie letztendlich eine Gesamtlösung ergeben.

Dieser Ansatz bietet sich prinzipiell auch bei der Entwicklung neuer Elementtypen an. So kann man eine komplexe Operation durch Verknüpfung einfacherer Operationen implementieren. Das neue Architekturelement für die komplexe Operation wäre dann ein Stellvertreter für die verknüpften Architekturelemente für die einfachen Operationen. Da die von ECL verwendete Architekturbeschreibungssprache Acme interne Architekturen in einzelnen Architekturelementen unterstützt, ist dieser Stellvertreteransatz auch leicht zu realisieren.

Wie in Kapitel 5.1 allerdings schon begründet wurde, besitzt ECL keinen allgemeinen rekursiven Elementtyp, der hierfür eingesetzt werden kann. Das Hauptargument besteht dabei darin, dass in der Anwendung seitens des Applikationsentwickler ein derartiges rekursives Element immer in einem bestimmten Kontext steht, der eine unterschiedliche Repräsentation und Behandlung erfordert. So sind z.B. Operationen und entfernte Dienstanbieter beide für den Stellvertreteransatz geeignet, aber dennoch beide sehr unterschiedlich.

Daher muss ein Sprachentwickler sich zuerst den Kontext überlegen, in dem er seinen neuen rekursiven Elementtyp definieren möchte. Konkret bedeutet dies, dass er sich für ein Erweiterungsmodul entscheiden muss, das einen Elementtyp enthält, welchen er als Supertyp für seinen neuen rekursiven Typ benutzt. Dieses kann dann z.B. ein Elementtyp für Operation sein.

Dann muss er sich entscheiden, mit welcher Semantik er seinen rekursiven Elementtyp versehen möchte. Im allgemeinen wird dies eine Substitutionssemantik sein, d.h. zur Übersetzungszeit wird das rekursive Element durch die interne Architektur ersetzt. Für diese Semantik existiert im ECL-System Unterstützung für Übersetzer und Editoren, die eine Implementierung stark

vereinfacht. Es sind aber auch andere Semantiken denkbar, die dann von dem Sprachentwickler implementiert werden müssen. Ein Beispiel hierfür ist die Unterstützung des Farm-Musters für nebenläufige Applikationen, das in Kapitel 9.4.2 beschrieben ist.

Wird eine Substitutionssemantik eingesetzt, müssen keine neuen Klassen für Applikationsobjekte implementiert werden, da nur die Klassen für die Applikationsobjekte der internen Architektur benutzt werden. Bei einer anderen Semantik ist allerdings eine entsprechende Applikationsobjektklasse zu implementieren, die diese Semantik unterstützt.

Spezialisierung vorhandener Elementtypen

Es lassen sich zwei Ansätze bei der Entwicklung von Code für größere Systeme unterscheiden:

- Der Code wird grob in wenige nach außen sichtbare Einheiten zerlegt. Diese einzelnen Einheiten besitzen eine umfangreiche Funktionalität und sind stark parametrisierbar. Abhängigkeiten zwischen den Einheiten ergeben sich nur explizit durch Nutzung ihrer Schnittstellen. Dieser Ansatz wird i.a. bei komponentenorientierter Softwareentwicklung verfolgt.

- Der Code wird sehr fein gegliedert, wobei zwischen den einzelnen Teilen starke Abhängigkeiten existieren. Dieser Methode wird häufig bei objektorientierter Softwareentwicklung eingesetzt, wobei die einzelnen Codeteile durch Klassen repräsentiert werden und die Abhängigkeiten, die mit dem Ziel der Wiederverwendung eingeführt werden, durch Vererbungsmechanismen realisiert sind.

Beide Ansätze haben Vor- und Nachteile [BS02]. Bei dem komponentenorientierten Ansatz ist zumindestens prinzipiell gewährleistet, dass keine versteckten Abhängigkeiten zwischen den einzelnen Komponenten existieren und dass die bekannten Abhängigkeiten klar spezifiziert sind. Dafür besitzen Komponenten häufig eine zu umfangreiche Funktionalität, die in der Applikation evtl. nicht komplett benötigt wird, und müssen über eine evtl. komplexe Parametrisierung an die jeweiligen Erfordernisse angepasst werden.

Dagegen stehen dem Anwendungsentwickler bei dem objektorientierten Ansatz eher spezialisierte Klassen zur Verfügung, die sich gut in den aktuellen Kontext integrieren lassen. Stehen diese nicht zur Verfügung, kann er über Vererbungsmechanismen spezialisierte Klassen schnell implementieren. Dafür besteht die Gefahr, dass sehr viele Abhängigkeiten eingeführt werden, welche die Wartbarkeit verringern und die Fehleranfälligkeit der Applikation erhöhen kann.

In dem hier vorgestellten System ECL, wird eine Applikation als Softwarearchitektur implementiert. Die Komponenten der Softwarearchitektur verweisen dabei i.a. auf externen Code, der zur Laufzeit in den Ablauf der Applikation integriert wird. Bei den meisten Applikationen wird der Großteil der Rechenzeit einer Applikation nicht in dem Koordinationssystem, sondern in dem externen Code verbraucht werden. Dies gilt insbesondere für rechenin-

tensive Applikationen. Wurde dieser externe Code nicht explizit für die jeweilige Applikation erstellt, sondern einer allgemeinen Bibliothek entnommen, wird damit der komponentenorientierte Ansatz verfolgt.

Die Kombination mit einer Architekturbeschreibungssprache ermöglicht es aber, dem Nachteil der erschwerten Handhabbarkeit aufgrund der komplexen Konfiguration des externen Codes entgegen zu wirken. Ein Sprachentwickler kann das Vokabular um neue Subtypen erweitern, die entsprechend vorkonfiguriert sind. Der Applikationsentwickler kann diese Subtypen ohne weitere Parametrisierung in seiner Applikation einsetzen.

Abbildung 5.8 enthält ein entsprechendes Beispiel in Form eines UML-Klassendiagramms. Der hypothetische Komponententyp `ExternalProgram` dient zur Ausführung eines externen Programms. Das Programm wird mit mit dem Attribut `program` spezifiziert, dessen Parameter mit dem Attribut `parameters`. In Kapitel 10.2 wird ein konkreter Komponententyp vorgestellt, der neben der Ausführung von Programmen auch ihre Übersetzung und Bindung unterstützt. Hier wird auf eine genaue Beschreibung des Komponententyps verzichtet und dieser nur zur Verdeutlichung der Erweiterung des Vokabulars durch neue Untertypen benutzt.

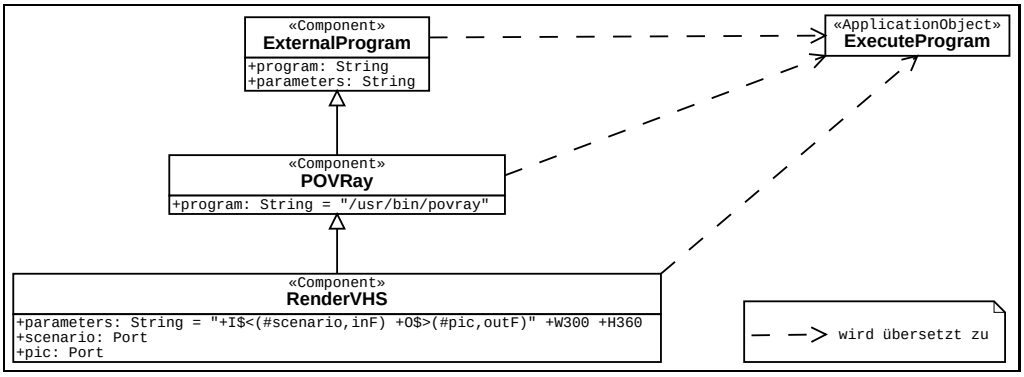

Abbildung 5.8.: Ein Beispiel für eine Erweiterung des Vokabulars durch Spezialisierung

Die Ausführung zur Laufzeit geschieht mittels eines Applikationsobjekts der Klasse `ExecuteProgram`. Dieses benutzt die beiden Attribute aus der Komponente, um das Programm zu lokalisieren und auszuführen. Indem man das `program` Attribut in dem Untertyp `POVRay` auf einen bestimmten Wert festlegt, erhält man einen neuen Komponententyp, der nur ein bestimmtes Programm ausführt. In diesem Fall ist es das Programm POV-Ray, ein Strahlenverfolgungsprogramm zur Erzeugung fotorealistischer Bilder.

Der Untertyp `RenderVHS` bietet eine weitere Vereinfachung für den Applikationsentwickler. Bei diesem ist das Attribut `parameters` zusätzlich vorbesetzt und er enthält zwei Anschlüsse. Neben dem Setzen der funktionalen Parameter, das hier aus der Spezifikation der gewünschten Auflösung auf die Werte der VHS-Norm besteht, wird auch eine Verbindung der Ein- und Ausgabedaten mit den Anschlüssen der Komponente definiert. Dabei wird der

Anschluss `scenario` für die zu berechnende Szene und der Anschluss `pic` für das resultierende Bild benutzt. Auf eine Beschreibung der Syntax wird hier verzichtet und auf das oben erwähnte Kapitel 10.2 verwiesen. Mit diesem Komponententyp `RenderVHS` hat der Applikationsentwickler nun ein Element zur Verfügung, das er ohne weitere Anpassung mit anderen Architekturelementen verknüpfen kann.

Bei dieser Art der Erweiterung des Vokabulars muss der Sprachentwickler im wesentlichen nur neue Typen in der Architekturbeschreibungssprache implementieren. Er muss keine neuen Klassen für Elementübersetzer oder Applikationsobjekte implementieren, da die Klassen für den Supertyp sowohl ausreichen als auch von dem ECL-System automatisch benutzt werden. Allerdings ist es empfehlenswert, wenn auch nicht zwingend notwendig, neue Elementeditoren und Repräsentationen zu implementieren. In dem obigen Beispiel sollte es z.B. dem Applikationsentwickler nicht möglich sein, bei Komponenten des Typs `RenderVHS` neue Anschlüsse oder neue Attribute hinzuzufügen. Desweiteren sollten sich diese Komponenten visuell deutlich von den allgemeinen `ExternalProgram`-Komponenten unterscheiden.

Integration neuer Architekturstile

Der dritte Ansatz für die Erstellung neuer Erweiterungsmodule ist der aufwendigste. Er besteht in der Entwicklung eines neuen Architekturstils, der Funktionalität enthält, die von keinem anderen Erweiterungsmodul angeboten wird. In diesem Fall muss der Sprachentwickler neue Klassen für Elementeditoren, Elementübersetzer und für Applikationsobjekte entwickeln und zu einem neuen Erweiterungsmodul bündeln. Eventuell kann dabei auf Klassen anderer Module aufgesetzt werden. Es wird aber immer notwendig sein, diese Klassen um neuen Code zu erweitern.

Um die Implementierung neuer Klassen für ein Erweiterungsmodul zu vereinfachen, stehen in dem ECL-System komplexe Basisklassen zur Verfügung, die den Großteil der notwendigen Verwaltungsarbeiten übernehmen. Der Sprachentwickler kann sich so auf die Implementierung der speziellen Funktionalität konzentrieren.

Möglicherweise ist es auch notwendig, dass der Komponentenentwickler Erweiterungen oder Anpassungen an dem Zielsystem vorzunehmen hat, um die gewünschte Funktionalität des neuen Erweiterungsmoduls realisieren zu können. In Kapitel 10.2 wird z.B. ein neuer Komponententyp beschrieben, der es ermöglicht, entfernt Programme zu übersetzen und auszuführen. Neben neuen Architekturelementtypen musste zusätzlich auch das Zielsystem so erweitert werden, dass der Zugriff auf diese entfernten Ressourcen ermöglicht wurde.

5.4. Interaktion unterschiedlicher Erweiterungsmodule

In dem vorherigen Kapitel wurde beschrieben, wie sich neue Erweiterungsmodule durch Kombination vorhandener Erweiterungsmodule entwickeln lassen. Für den Applikationsentwickler stellt sich aber auch die Frage, wie er in

einer Applikationsarchitektur verschiedene Erweiterungsmodule kombiniert benutzen kann.

Die verschiedenen Optionen für die Realisierung einer kombinierten Nutzung sollen in diesem Kapitel an einem einfachen Beispiel motiviert und vorgestellt werden. Der Anwendungsentwickler möchte sowohl Dienstanbieter ansprechen, die eine CORBA-Schnittstelle besitzen, als auch Dienstanbieter, die über eine SOAP-Schnittstelle verfügen. Er wird also für seine Applikationsarchitektur sowohl ein CORBA- als auch ein SOAP-Erweiterungsmodul wählen und die zugehörigen Architekturelemente miteinander verbinden wollen.

Bei einer allgemeinen Betrachtung der Problematik zeigt sich, dass nicht immer zu gewährleisten ist, dass die Vokabulare zweier beliebiger Erweiterungsmodule miteinander kompatibel sind. Schließlich kann ein Erweiterungsmodul ein ereignisbasiertes Kommunikationsparadigma unterstützen, während das andere auf einer deklarativen Spezifikation von Abhängigkeiten zwischen den Komponenten basiert. Hier ist es absolut unklar, welche Semantik mit der Komposition von Architekturelementen beider Vokabulare assoziiert sein soll.

Die Hauptursache für diese prinzipielle Inkompatibilität zwischen zwei beliebigen Erweiterungsmodulen liegt darin begründet, dass es in ECL kein allgemeines semantisches Modell gibt, welches für die einzelnen Architekturelemente eingesetzt wird. Der kleinste gemeinsame Nenner ist die Verwendung der Architekturbeschreibungssprache für die Syntax der Vokabulare. Diese ist aber für eine kompatible Verknüpfung nicht ausreichend.

Trotzdem ist an dem Beispiel zu erkennen, dass eine Komposition von Architekturelementen aus unterschiedlichen Vokabularen dann sinnvoll ist, wenn sie eine *ähnliche* Semantik besitzen. Im allgemeinen sind sie dann ähnlich, wenn sie dem gleichen Koordinationsparadigma unterliegen. In dem Beispiel besteht das Paradigma aus einem Operationenaufruf bei einem Dienstanbieter, wobei allerdings unterschiedliche und zusätzlich inkompatible Middleware-Systeme eingesetzt wird. Es gibt zwei Ansätze in dem ECL-System, wie Architekturelemente aus unterschiedlichen Erweiterungsmodulen miteinander interagieren können:

1. Interaktion über gemeinsame Basisklassen eines Basismodells,
2. Interaktion über Mediatoren aus vereinigenden Unterklassen.

Gemeinsame Basisklassen

In dem ersten Fall enthalten zwei Erweiterungsmodule B und C Elementtypen T_B und T_C, die beide von dem gleichen Basistyp T_A aus einem Erweiterungsmodul A abgeleitet sind. Dabei kann auch $A = B$ oder $A = C$ gelten. Bezogen auf das Beispiel kann A Architekturelemente enthalten, die entfernte Dienstanbieter und entsprechende Klienten darstellen. Diese Elemente können abstrakt definiert sein und mit keiner konkreten Funktionalität für ein bestimmtes Zielsystem verknüpft sein. B enthält dann konkrete Architekturelemente für das CORBA-System und C Elemente für SOAP. Wenn man nun dafür sorgt, dass die Applikationsobjekte von B mit den Applikationsobjekten von C kompatibel sind, indem sie z.B. die gleiche Basisklasse benutzen, dann lassen sich

die zugehörigen Architekturelemente in der Applikationsarchitektur miteinander kombinieren.

Kapitel 6 enthält ein anderes Beispiel für diesen Ansatz. Dort wird ein Erweiterungsmodul vorgestellt, das es ermöglicht, eine Applikationsarchitektur als nichtsequenziellen Ablauf von abstrakten Operationen zu implementieren. Andere Erweiterungsmodule setzen auf diesem Modul auf und bieten Komponententypen für konkrete Operationen an. Da die Applikationsobjekte für diese konkreten Operationen alle auf der gleichen Basisklasse aufsetzen, sind sie miteinander kompatibel.

Mediatoren

Bei dem zweiten Ansatz wird für zwei Erweiterungsmodule A und B ein neues Erweiterungsmodul C implementiert, welches Architekturelemente enthält, mit denen sich Elemente aus A mit Elementen aus B kombinieren lassen und umgekehrt. Auf das Eingangsbeispiel bezogen, enthält A z.B. einen Konnektortyp cn_A zur Verbindung von CORBA-Klienten zu CORBA-Dienstanbietern. B enthält entsprechend für SOAP einen Konnektortyp cn_B. Das Modul C enthält dann einen Konnektortyp cn_C, der sowohl von cn_A als auch von cn_B abgeleitet ist und der eine beliebige Verbindung von Klienten zu Dienstanbieten beider Middleware-Systeme unterstützt.

Beide Ansätze haben unterschiedliche Vor- und Nachteile. Bei der Interaktion über gemeinsame Basisklassen muss ein allgemeines semantisches Modell definiert werden, an dem sich alle aufsetzenden Erweiterungsmodule orientieren müssen. Sie werden dadurch bzgl. der Gestaltungsfreiheit für die einzelnen Architekturelemente begrenzt, so dass evtl. das Zielsystem nicht komplett unterstützt werden kann oder die Architekturelemente unnötig kompliziert sind.

Als Beispiel, wie ein zu einfaches Basismodell darauf aufsetzende Erweiterungsmodule beschränken kann, sei ein Erweiterungsmodul gegeben, das auf einem abstrakten Modul aufsetzt, welches entfernte Dienstanbieter unterstützt, die über feste Schnittstellen aufrufbar sind. Das neue Modul möchte dem Applikationsentwickler nun Hilfsmittel zur Verfügung stellen, um entfernte Dienstanbieter unter Einsatz des Fabrik-Entwurfsmusters [GHJ97] dynamisch zu erzeugen. Dafür führt das Modul einen neuen Konnektortyp ein, der eine Komponente für den erzeugten Dienstanbieter S mit der Komponente F für die Fabrik verbindet. Die Semantik besteht darin, dass beim Start der Applikation über die Fabrik F ein Dienstanbieter S erzeugt wird.

Dieser Konnektortyp ist aber von dem Modul neu eingeführt worden und basiert nicht auf dem allgemeinen Basismodell. Er ist damit mit Architekturelementen aus anderen Erweiterungsmodulen, die auf diesem Modell aufsetzen, inkompatibel.

Alternativ lässt sich das Fabrik-Entwurfsmuster auch dadurch implementieren, dass der Klient die Fabrik direkt anspricht. Der dynamisch erzeugte Dienstanbieter ist dann in der Applikationsarchitektur nicht sichtbar. Die Kompatibilität ist damit zwar gewährleistet, dafür ist die Applikationsarchitektur nicht mehr vollständig und es besteht die Gefahr, dass verdeckte

Abhängigkeiten eingeführt werden. Schließlich kann der unsichtbare, dynamisch erzeugte Dienstanbieter auf andere Dienstanbieter zugreifen, die über Komponenten in der Applikationsarchitektur repräsentiert werden.

Die bei dem zweiten Ansatz verwendeten Mediatoren dagegen sind höchst flexibel, da sie keinen Beschränkungen eines allgemeinen Modells unterworfen sind. Bei ihrer Entwicklung haben sowohl Sprach- als auch Komponentenentwickler freie Hand, wie sie die Inkompatibilität zwischen zwei Architekturelementen überbrücken können. Stehen dabei mehrere Alternativen zur Verfügung können die besten gewählt werden.

Bei Mediatoren zwischen zwei Systemen mit inkompatiblen Typen hat man z.B. die Wahl, ob man das Typsystem eines Systems auf Kosten von evtl. Informationsverlust bei Daten des anderen Systems als das primäre Typsystem benutzt, oder ob man ein neues Typsystem einführt, das den kleinsten gemeinsamen Nenner beider Typsysteme realisiert. Für alle drei Alternativen lassen sich Mediatoren entwickeln.

Der größte Nachteil des Mediatoransatzes besteht in dem hohen Arbeitsaufwand für den Sprachentwickler, da er für prinzipiell alle sinnvollen Kombinationen inkompatibler Architekturelemente spezielle Mediatoren entwickeln muss. Zusätzlich muss der Applikationsentwickler in der Lage sein, die benötigten Mediatoren auch finden und benutzen zu können.

Es ist schwierig, einen dieser beiden Ansätze, Basismodell und Mediatoren, als dem anderen deutlich überlegen zu bewerten. Fallstudien haben allerdings gezeigt, dass sich für Architekturelemente, die sich sinnvoll miteinander verbinden lassen, ein Basismodell stets finden lässt. In Kapitel 14 werden Richtlinien für den Entwurf derartiger Basismodelle vorgestellt.

Die an dem Beispiel des Fabrik-Entwurfsmusters beschriebene Problematik, dass die Anwendungsklasse die Unterstützung weiterer Koordinationsparadigmen erfordert oder dass nicht alle Fähigkeiten der Zielsysteme genutzt werden können, ist mit spezialisierten zusätzlichen Architekturelementtypen lösbar. Die entstehende Inkompatibilität ist entweder tolerierbar oder lässt sich mit Mediatoren überwinden. Zusammengefasst ist also eine Kombination beider Ansätze empfehlenswert.

Damit sichergestellt ist, dass der Applikationsentwickler nur Architekturelemente miteinander verbindet, die kompatibel sind, enthält ein Vokabular neben den Typinformationen auch logische Regeln, die kompatible Verknüpfungen definieren. Hierfür bietet die eingesetzte Architekturbeschreibungssprache die in Kapitel 5.2.1 vorgestellte Möglichkeit, Invarianten über zulässige Architekturen zu spezifizieren und automatisch zu überprüfen.

Abbildung 5.9 zeigt für das Eingangsbeispiel Vokabulare mit ihren Invarianten. Der Stil `CORBA` enthält Komponententypen für Dienstanbieter und Klienten und einen Konnektortyp, um beide zu verbinden. Die Invariante besagt, dass `CORBAServer`-Komponenten nur mit `CORBAConn`-Konnektoren verknüpft werden können. Auf eine entsprechende Invariante für `CORBAClient`-Komponenten wurde hier verzichtet. Der Stil `SOAP` ist entsprechend für SOAP realisiert.

Der Stil `SOAP_CORBA_Bridge` enthält einen Konnektortyp

```
Style CORBA extends ECL = {
  Component Type CORBAServer = {...};
  Component Type CORBAClient  =  {...};
  Connector Type  CORBAConn = {..};

  invariant forall serv in
   { select cp in self.components |
       declaresType(cp, CORBAServer)}
   | forall cn in self.connectors
     | (attached(serv, cn) -> declaresType(cn, CORBAConn);
  ...
};

Style SOAP extends ECL = {
  Component Type SOAPServer = {...};
  Component Type SOAPClient  =  {...};
  Connector Type  SOAPConn = {..};

  invariant forall serv in
   { select cp in self.components |
       declaresType(cp, SOAPServer)}
   | forall cn in self.connectors
     | (attached(serv, cn) -> declaresType(cn, SOAPConn);
  ...
};

Style SOAP_CORBA_Bridge extends CORBA, SOAP with {
  Connector Type SOAP_CORBA_Conn
    extends CORBAConn, SOAPConn with {...};
};
```

Abbildung 5.9.: Syntaktische Spezifikation kompatibler Verknüpfungen in Acme/Armani

SOAP_CORBA_Conn, der von den Konnektortypen der beiden anderen Stile abgeleitet ist. Mit Konnektoren dieses Typs ist es möglich, CORBAServer-Komponenten mit SOAPClient-Komponenten zu verknüpfen, ohne die Invarianten zu verletzen. Die Kompatibilität lässt sich also mit diesem Ansatz auf der syntaktischen Ebene ausdrücken.

Allerdings ist zu beachten, dass die Invarianten sorgfältig so zu formulieren sind, dass spätere Erweiterungen möglich sind. Angenommen der CORBA-Stil würde z.B. eine Invariante besitzen, die besagt, dass CORBAServer-Komponenten immer mit CORBAClient-Komponenten zu verknüpfen sind. Dann würde der Ansatz mit einem SOAP_CORBA_Bridge-Konnektortyp nicht funktionieren. Alternativ wäre statt dessen ein Komponententyp denkbar, der von CORBAClient und SOAPClient abgeleitet ist. Der Applikationsentwickler muss dann immer den neuen Komponententyp für Klienten-Komponenten benutzen.

5.5. Analyse von Applikationen

Damit Applikationen in dem ECL-System analysierbar sind, müssen alle für die Analyse notwendigen Abhängigkeiten in der Applikationsarchitektur explizit enthalten sein, bzw. sich aus den expliziten Abhängigkeiten erschließen lassen. Dass dieses möglich ist, liegt in der Verantwortung des Entwicklers des Erweiterungsmoduls und stellt damit eine Designentscheidung dar.

Ein typischer Ansatz zur Applikationsanalyse besteht darin, die einzelnen Elementtypen mit Analyseinformationen, wie. z.B. Verhaltensbeschreibungen, zu annotieren. Ein Analysewerkzeug enthält nun deduktive Regeln, mit denen es möglich ist, eine Applikationsarchitektur durch Verknüpfung der einzelnen Elemente zu analysieren. Ein Machbarkeitsbeweis für diesen Ansatz wurde in [KF00] gegeben. Dort wurde das Verhalten der einzelnen Elementtypen mit Prozessalgebren spezifiziert. Unter Einsatz einfacher Kompositionsregeln wurde das Gesamtverhalten der Applikation aus den einzelnen Elementen abgeleitet.

5.6. Implementierung von ECL

Das Grundkonzept von dem erweiterbaren Koordinationssprachensystem ECL besteht darin, dass der Applikationsentwickler sich für seine Applikation zuerst eine Menge von Erweiterungsmodulen auswählt, welche alle für seine Anwendung benötigten Elemente enthalten. Durch Komposition dieser Elemente erstellt er seine Applikation. Diese wird schließlich übersetzt und von einem modularen Laufzeitsystem ausgeführt.

Die Architektur der Implementierung von ECL besteht daher aus den Erweiterungsmodulen mit ihren Elementen für Applikationen, Übersetzern und Laufzeitunterstützungen und einer grafischen Benutzerschnittstelle. Letztere dient zur Verwaltung der lokal installierten Erweiterungsmodule und dem interaktiven Erstellen von Anwendungen. Da zur Übersetzungszeit der Benutzerschnittstelle i.a. nicht alle Erweiterungsmodule vorliegen, muss sie in

der Lage sein, diese zur Laufzeit dynamisch nachzuladen. Zusätzlich müssen die Abhängigkeiten zwischen den Erweiterungsmodulen ebenfall dynamisch aufgelöst werden. Dazu gehört z.B., dass Klassen eines Erweiterungsmodul Spezialisierungen von Klassen eines anderen Moduls sein können und daher Zugriff auf den entsprechenden Code benötigen. Aufgrund dieser Anforderungen bzgl. Modularität und Dynamik basiert ECL auf dem Komponentensystem *Open Services Gateway Initiative (OSGi)* [Gon01, osg00], das einen Rahmen für eine dynamische Installation und Konfiguration von Codefragmenten bietet.

Übersetzte Applikationen werden von der Benutzerschnittstelle in einer Datei gespeichert. Diese Datei enthält die Applikationsobjekte und die Laufzeitunterstützungen, die von den eingesetzten Erweiterungsmodulen bereitgestellt werden. Mit einem Starterprogramm wird die Anwendung dann in einer passenden Umgebung gestartet. Wenn z.B. die Anwendung Elemente enthält, die einen Zugriff auf entfernte Datenobjekte ermöglichen, dann müssen diese Datenobjekte und ihre zugehörige Infrastruktur auch beim Starten der Applikation vorhanden sein. Die in der Datei enthaltenen Laufzeitmodule ermöglichen nur einen Zugriff auf die Umgebung.

Die folgenden Kapitel stellen die Implementierung detailliert vor. Dafür werden zunächst die eingesetzten Technologien vorgestellt. Dann wird die allgemeine Architektur der Implementierungen der Erweiterungsmodule präsentiert. Abschließend wird die grafische Benutzerschnittstelle erklärt.

5.6.1. Eingesetzte Technologien

Als Programmiersprache für ECL wurde Java gewählt, da diese eine ausreichende Effizienz bei einer großen Anzahl leistungsfähiger Bibliotheken und einem stabilen Laufzeitsystem bietet und von vielen Plattformen unterstützt wird. Um die Anforderungen von ECL bzgl. einem dynamischen Nachladen von Erweiterungsmodulen zu erfüllen, wurde das Komponentensystem Oscar eingesetzt [osc02]. Oscar ist eine frei verfügbare, partielle Implementierung der Spezifikationen der OSGi [Gon01, osg00]. Die OSGi wurde 1999 gegründet mit dem Ziel, ein Java-basiertes System zu spezifizieren, das die Nutzung von Diensten in lokalen Netzen ermöglicht, die wiederum über Weitverkehrsnetzen zur Verfügung gestellt werden. Ein konkretes Anwendungsgebiet, auf das OSGi zielt, ist die Steuerung von Kleingeräten in hausinternen Netzen.

OSGi bezeichnet sich selbst nicht als ein Komponentensystem. Eine Sammlung von Ressourcen, die sowohl Code als auch Daten umfassen kann, wird in OSGi *Bündel* genannt. Ein Bündel exportiert Dienste und Ressourcen, die in Java-Paketen zusammengefasst werden, an andere Bündel. Ein Bündel kann statisch[3] von anderen Bündeln abhängen, indem es Dienste oder Ressourcenpakete importiert. Die Auflösung dieser Abhängigkeiten geschieht von dem OSGi-Laufzeitsystem, wenn ein Bündel aktiviert und damit in das System integriert wird. Jedes Bündel besitzt einen Ursprung in Form einer URL. Diese

[3]Mit statischer Abhängigkeit ist hier Abhängigkeit zur Installations- bzw. Aktivierungszeit gemeint. Diese Abhängigkeiten sind in einem Dokument in jedem Bündel explizit formuliert.

ermöglicht es, Bündel automatisch zu aktualisieren. Jedes Erweiterungsmodul in ECL und auch die Benutzerschnittstelle ist in einem OSGi-Bündel verpackt. Auch die eingesetzten Bibliotheken sind als OSGi-Bündel realisiert.

Die Implementierung der interaktiven Bearbeitung von Applikationsarchitekturen setzt auf der Bibliothek GEF auf [gef02]. Diese basiert auf dem Model-View-Controller Entwurfsmuster und unterstützt die Darstellung und Bearbeitung von Graphen. Für jedes Architekturelement der Erweiterungsmodule existieren spezielle Modell-, Präsentations- und Bearbeitungsklassen, die auf den GEF-Klassen aufbauen.

Applikationsarchitekturen werden intern mit der Architekturbeschreibungssprache Armani repräsentiert [Mon01]. Für Armani ist eine in Java geschriebene Bibliothek verfügbar, die es ermöglicht, aus Architekturbeschreibungen in Armani ein internes Modell zu erstellen, dieses zu bearbeiten und wieder in Textform zu speichern. Diese Bibliothek ist allerdings nicht frei verfügbar, sondern wurde von ihrem Autor nach persönlicher Absprache zur Verfügung gestellt.

5.6.2. Erweiterungsmodule

Alle Erweiterungsmodule basieren auf einem Basismodul, das die grundlegende Funktionalität für die einzelnen Elemente und den Übersetzungsmechanismus bereitstellt. Abbildung 5.10 skizziert die wichtigsten Klassen und Schnittstellen für ein Erweiterungsmodul, die in dem Basismodul zur Verfügung gestellt werden[4]. Jedes Erweiterungsmodul hat einen eindeutigen Namen. Das von ihm bereitgestellte Architekturvokabular besteht aus einer textuellen Spezifikation in Armani. Die Namen der Module, auf die das Erweiterungsmodul aufsetzt, werden von `getDependencies()` geliefert.

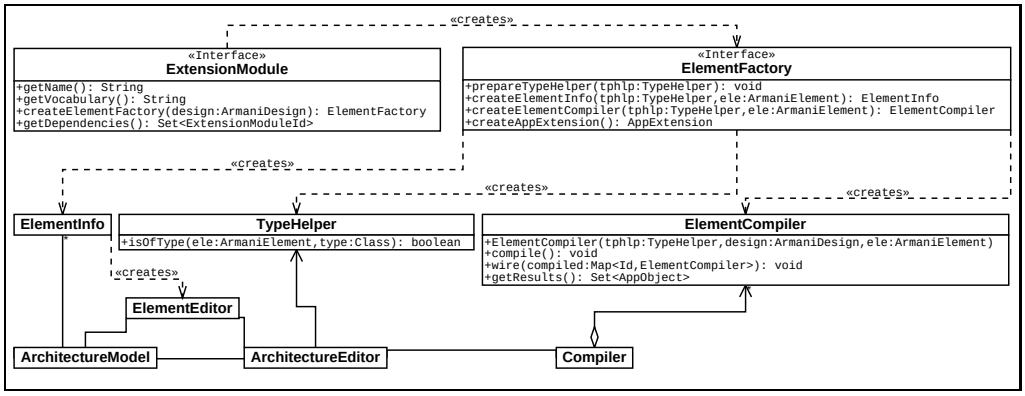

Abbildung 5.10.: Diagramm der wichtigsten Klassen für ein Erweiterungsmodul

[4]Um das Verständnis zu erleichtern, wurden die hier vorgestellten Klassendiagramme vereinfacht. Insbesondere wurden Klassen zusammengefasst und die Namensgebung angepasst. Desweiteren wurden bei den Signaturen von Behältertypen polymorphe Typen eingesetzt, die Java nicht bietet. Die konkrete Implementierung ist deutlich komplexer und die benutzten Namen sind aufgrund des evolutionären Entwicklungsansatzes nicht durchgehend konsistent.

Objekte von Elementeditorklassen und Elementübersetzerklassen, im folgenden Hilfsobjekte genannt, werden mit einer `ElementFactory` erzeugt. Sie wird von dem `ExtensionModule` für einen gegebenen Architekturentwurf[5] erzeugt. Um für ein Architekturelement die richtigen Hilfsobjekte zu erzeugen, wird auf dessen Armanityp zurückgegriffen. Eine Elementfabrik unterstützt dabei nur eine Teilmenge der insgesamt der Applikation zur Verfügung stehenden Elementtypen. Kann sie für ein gegebenes Architekturelement kein Hilfsobjekt erzeugen, gibt sie `null` zurück.

Um ein Hilfsobjekt für ein beliebiges Architekturelement zu erzeugen, benutzt man den mit der Applikation assoziierten Architektureditor, ein Objekt der Klasse `ArchitectureEditor`. Dieser kennt alle verfügbaren Elementfabriken und eine korrekte Abfragereihenfolge[6]. Zur Erzeugung des Hilfsobjekts geht der Architektureditor alle Elementfabriken solange durch, bis er eine gefunden hat, die ein Hilfsobjekt erzeugen kann. Der Architektureditor fügt sich zusätzlich als Verwalter eines Applikationsgraphens in das GEF-System ein, welches für die grafische Bearbeitung von Applikationen eingesetzt wird.

Zur Vereinfachung des Umgangs mit den Armanitypen dient die Klasse `TypeHelper`. Sie besitzt intern eine Abbildung von allen Armanitypen in einem Architekturentwurf auf Java-Klassen, welche dann in der Java Implementierung zur Abfrage des Typs benutzt wird. Die Methode `isOfType(ele,cls)` liefert genau dann `true`, wenn der Armanityp von `ele` der mit der Klasse `cls` assoziierte Typ oder ein abgeleiteter Typ ist. Als geeignete Java-Klasse bietet sich die dem Architekturtyp zugeordnete `ElementInfo` Klasse an, welche die interne Präsentation der Architekturelemente in dem GEF-System darstellt.

Ein Objekt `tphlp` der Klasse `TypeHelper` wird initialisiert, in dem jede der Applikation zugeordnete Elementfabrik sie durch `prepareTypeHelper(tphlp)` mit den von ihr unterstützten Elementtypen initialisiert. Mit diesem Ansatz werden die textuelle Spezifikation der Armanitypen von der Implementierung der Hilfsobjekte entkoppelt. Eine Änderung der Typnamen resultiert nur in einer Anpassung der Elementfabrik.

Objekte von Unterklassen von `ElementInfo` bieten Zugriff auf alle Eigenschaften eines Architekturelements, die für die Hilfsobjekte relevant sind. Sie sind zudem in der Lage, für sich Elementeditoren[7] zu erzeugen, die dann von dem Applikationsentwickler benutzt werden können, um das Element zu bearbeiten.

Der Zusammenhang zwischen `Compiler` und `ElementCompiler` Objekten und ihre Funktionsweise wurde schon in Kapitel 5.2.2 detailliert beschrieben. Für die Visualisierung und die interaktive Manipulation von Architek-

[5] Ein Architekturentwurf besteht aus der Applikationsarchitektur selber und dem ihr zur Verfügung stehenden Vokabular.

[6] Die Bestimmung dieser Reihenfolge ist in Kapitel 5.2.1 beschrieben.

[7] Die oben angesprochene Vereinfachung bei der Präsentation der Implementierung ist bei den Elementeditoren am stärksten ausgeprägt. Diese beinhalten sowohl eine grafische Darstellung der Architekturelemente als Graphkomponenten unter Einsatz der GEF Bibliothek als auch weitere grafische Schnittstellen unter Verwendung der Swing Bibliothek. Daher ist die tatsächliche Klassenstruktur recht komplex und aufwändig. Da sie aber keinen besonders lohnenden Einblick in die allgemeine Struktur von ECL bietet, wurde hier stark abstrahiert.

turen wird das Modell-View-Controller Entwurfsmuster eingesetzt. Als Modell fungiert die Klasse `ArchitectureModel`, welche die Armani Architektur kapselt. Der Verwalter ist der Architektureditor. Für die Präsentation werden die Elementeditoren eingesetzt. Alle drei Klassen setzen auf in der GEF Bibliothek zur Verfügung gestellten Klassen auf.

5.6.3. Benutzerschnittstelle

Die Implementierung der Benutzerschnittstelle gliedert sich in zwei Teile: eine Java-Applikation zum Starten des Systems und ein OSGi-Bündel, welches die interaktive Funktionalität enthält. Der Einsatz eines Bündels hat den Vorteil, dass die benötigten Bibliotheken ebenfalls als Bündel verwaltet werden können und damit nicht in der Systemumgebung des Entwicklers installiert werden müssen.

Die Starteranwendung erzeugt zuerst ein Exemplar einer OSGi-Umgebung. Bei dem ersten Aufruf werden die notwendigen Bündel installiert. Dazu gehören die Bibliotheken, der interaktive Teil der Benutzerschnittstelle und das Basiserweiterungsmodul. Andernfalls werden alle installierten Bündel aktiviert. Zusätzlich wird ein spezielles Bündel aufgesetzt, welches es anderen Bündeln erlaubt, die OSGi-Umgebung zu terminieren.

Wenn das Bündel mit der Benutzerschnittstelle aktiviert wird, erzeugt es ein Fenster mit einer Menüleiste, über das der Anwender auf die Funktionen zugreifen kann. Für eine Beschreibung der Funktionen und für Abbildungen der grafischen Schnittstelle siehe Kapitel 5.

Für jede Operation, die dem Benutzer angeboten wird, existiert eine spezielle Kommandoklasse. Eine Kommandoklasse ist ein Konzept von GEF. Objekte solcher Klassen können grafischen Interaktionselementen, wie Knöpfe oder Menüs, als Aktionen übergeben werden. Bei Aktivierung greifen sie auf den Aktivierungskontext zu, in dem sich auch ein Verweis auf den Verwalter der gerade bearbeiteten Applikationsarchitektur befindet. Dieser Ansatz vermeidet die Aufnahme von Applikationslogik in der Präsentationsschicht, indem in dem Code für die Aktionen der Präsentationschicht, kein direkter Zugriff auf das Modell oder den Verwalter stattfindet.

5.7. Verwandte Arbeiten

Die in Kapitel 3 beschriebenen erweiterbaren Systeme basieren weitgehend auf einem bestimmten Basismodell mit einem festen Koordinationsparadigma. Damit eignen sie sich nicht als Grundlage, um Koordinationssysteme für beliebige Applikationsbereiche und Zielplattformen zu entwickeln.

Eine Ausnahme stellt die Architekturbeschreibungssprache UniCon dar, für die für unterschiedliche Anwendungsfelder spezialisierte Vokabulare entwickelt wurden [SDK+95]. Ein Applikationsentwickler entscheidet sich für ein Vokabular, erstellt die Applikationsarchitektur und übersetzt diese zu einem ausführbaren Programm. Es existiert allerdings kein allgemeiner Rahmen, welcher die Erstellung solcher Vokabulare mit einem zugehörigen Überset-

zer vereinfacht. Auch ist es nicht möglich, unterschiedliche Vokabulare zu mischen, oder ein neues Vokabular auf einem bestehenden aufzusetzen. UniCon stellt damit konzeptionell einen Vorgänger für ECL dar.

Ptolemy-II

Das Modellierungs- und Entwicklungssystem *Ptolemy II* wurde für den Applikationsbereich der eingebetteten Systeme entwickelt [LLEL02, HLL+02]. Eine Applikation besteht aus miteinander verknüpften Komponenten, die von einem Direktor koordiniert werden. Die Applikation wird in Ptolemy II als Modell bezeichnet, die Komponenten als Aktoren. Um ein einheitliches Vokabular zu benutzen, wird diese Terminologie im folgenden aber nicht eingesetzt.

Komponenten werden stets direkt über Anschlüsse miteinander verbunden. Es gibt keine unterschiedlichen Konnektortypen. Die Semantik einer Verknüpfung zweier Komponenten bestimmt der Direktor. Er definiert damit einen Architekturstil, der in Ptolemy II als Domäne bezeichnet wird.

Komponenten können selbst wiederum hierarchisch aufgebaut sein und beliebige Direktoren benutzen. Dies ermöglicht es, heterogene Applikationen unter Einsatz unterschiedlicher Architekturstile zu entwickeln. Interoperabilität zwischen den Komponenten wird über eine feste Schnittstelle der Direktoren ermöglicht. Diese Schnittstelle basiert auf einem Operationenparadigma und enthält Methoden wie z.B. `fire()` und `iterate()`.

Eine Applikation wird in der auf XML basierenden Beschreibungssprache *MoML* (Modeling Markup Language) implementiert. MoML unterstützt Basistypen für Komponenten, die mit Referenzen auf Java-Klassen parametrisierbar sind. Das Verhalten einer Komponente z.B. beim Empfang von Aktivierungen lässt sich bestimmen, indem die Java-Klasse mit dem gewünschten Verhalten in der MoML-Beschreibung gesetzt wird. MoML bietet auch die Möglichkeit über einfache Vererbung neue Komponententypen zu definieren. Diese unterscheiden sich von ihren Obertypen durch eine unterschiedliche Parametrisierung.

Damit ist die Beschreibungssprache von Ptolemy II, ebenso wie in ECL, frei von Semantik. Diese wird erst über Architekturstile (Domänen) und über Code, der von dem Applikationsentwickler zur Verfügung gestellt wird, dem System hinzugefügt. Allerdings ist MoML konzeptionell strikter als der in ECL gewählte Ansatz. Die Erweiterbarkeit durch Java-Klassen ist nur an bestimmten festen Punkten möglich. Daher kann jede in MoML implementierte Applikation von einem allgemeinen Interpreter sofort ausgeführt werden, sofern die referenzierten Java-Klassen vorliegen.

In ECL ist es dagegen notwendig, eine Applikation zuerst in ein ausführbares Format zu übersetzen. Dies bedeutet damit zwar einen zusätzlichen Aufwand bei der Entwicklung, bietet aber auch Potenzial für eine höhere Effizienz der Applikation. Eine Analyse zur Übersetzungszeit ermöglicht es, eine Applikation auf die für die Ausführung relevanten Informationen in einem effizient zugreifbaren Format zu reduzieren. In Kapitel 11.3 wird dies für ECL an einem Beispiel demonstriert.

Ptolemy II erlaubt es nicht, einen neuen Architekturstil auf vorhandene Architekturstile aufzusetzen. Auch wenn die Software-Architektur von

Ptolemy II dies prinzipiell ermöglicht, ist in dem konkreten System keine Unterstützung vorhanden. Hinzukommend erscheint die Vermischung von MoML-Klassen in XML und die Referenzierung von Java-Klassen im Vergleich mit der systematischen Trennung in ECL die Erstellung und Erweiterung von Architekturstilen zu erschweren.

Modellgetriebene Architekturen

Die Zielsetzung vom ECL-System besteht darin, unter Einsatz eines modular erweiterbaren Rahmens mit wenig Aufwand Entwicklungswerkzeuge zur Erstellung verteilter Anwendungen für beliebige Applikationsbereiche und für beliebige Zielplattformen bereitzustellen. Die gleiche Zielsetzung verfolgt die Object Management Group (OMG) mit ihrem Konzept für modellgetriebene Architekturen (MDA, *Model Driven Architecture*), wobei sie aber nicht bei den Entwicklungswerkzeugen sondern bei der allgemeinen Entwicklungsmethodik ansetzt [BO01, SG00].

Das zentrale Konzept in MDA ist das Modell. Ein Modell ist eine Repräsentation eines Ausschnitts der Funktion, der Struktur oder des Verhaltens eines Gesamtsystems. Die Spezifikationssprache für ein Modell wird wiederum mit einem Modell beschrieben, welches damit ein Metamodell darstellt. Als Basismodell in MDA wird die *Meta Object Facility* (MOF) benutzt [OMG02b]. Da diese aufgrund ihrer starken Generizität etwas unhandlich ist, umfasst MDA zwei weitere Basismodelle, die in MOF spezifiziert sind: *Unified Modeling Language* (UML) [OMG01b] und das *Common Warehouse Metamodel* (CWM) [OMG01a]. UML dient zur graphischen Spezifikation diskreter Systeme und wird i.a. für eine objektorientierte Spezifikation von Applikationen eingesetzt. CWM wurde konzipiert für die Beschreibung von Datenbank-basierten Applikationen und bietet Möglichkeiten zur Spezifikation von Daten und Metadaten.

Als grundlegende Entwicklungsmethode schlägt MDA einen Ansatz vor, in dem abstrakte Modelle schrittweise zu konkreteren Modellen transformiert werden. In dem ersten Schritt werden mehrere plattformunabhängige Modelle (PIM, Platform Independent Model) erstellt, die unterschiedliche Aspekte der Applikation modellieren. Diese werden schrittweise verfeinert. Eine Verfeinerung ist dabei eine Transformation von einem PIM zu einem anderen. Die Transformation selber wird wiederum durch ein Modell spezifiziert.

Im Laufe der Entwicklung kommt es dann zu einem Punkt, an dem die plattformunabhängigen Modelle auf plattformspezifische Modelle (PSM, Platform Specific Model) abgebildet werden. Diese werden wiederum schrittweise verfeinert, bis sie entweder eine Implementierung des Systems darstellen oder sich eine Implementierung direkt ableiten lässt.

Der Ansatz der schrittweisen Verfeinerung ist ein üblicher Ansatz in der Softwareentwicklung. Der Vorteil bei dem Einsatz von MDA besteht darin, dass in jeder Entwicklungsphase miteinander kompatible Spezifikationssprachen eingesetzt werden und dass der Zusammenhang zwischen den einzelnen Spezifikationsdokumenten in den unterschiedlichen Entwicklungsphasen durch Modelle exakt beschrieben ist. Dieser Vorteil ist dann besonders groß, wenn die einzelnen Modelle nicht nur in ihrer Syntax sondern auch in ihrer

Semantik genau spezifiziert sind. In diesem Fall ist es potenziell möglich, die einzelnen Spezifikationsdokumente automatisch bzgl. bestimmter Kriterien, wie z.B. Leistungsanforderungen, zu überprüfen und die Transformation zwischen unterschiedlichen Modellen automatisch oder zumindestens werkzeuggestützt durchzuführen.

Um unterschiedliche Applikationsbereiche und Zielplattformen einfach zu unterstützen, wird in dem MDA-Ansatz empfohlen, UML-Profile zu benutzen. Diese bestehen aus einer Menge von Stereotypen und benannten Werten mit einer Spezifikation ihrer Semantik. Die OMG stellt einige Profile bereit, wie z.B. ein Profil für CORBA-basierte Applikationen [OMG02d] oder ein Profil für die Integration von Geschäftsanwendungen [OMG02c].

Die Forschung bzgl. des konkreten Einsatzes von MDA befindet sich noch im Anfangsstadium. In [Ger02] wird versucht, die einzelnen Konzepte und ihre Zusammenhänge von MDA mit einem einfachen formalen Modell zu beschreiben, um so die vorhandenen Möglichkeiten und die Gemeinsamkeiten mit vorhandenen Systemen auszuloten. Es werden mehrere Bereiche identifiziert, die weitere Untersuchungen benötigen.

In [BG01] wird ein Softwareentwicklungsprozess vorgestellt, der auf MDA-Konzepten basiert. Er beschränkt sich dabei auf die Applikationen, die sich für den Einsatz des ISO-Referenzmodells für verteiltes offenes Rechnen (RM-ODP [ISO98]) eignen. Der Prozess besteht aus dem Erstellen eines plattformunabhängigen Modells, welches mit Hilfe eines Modells der Zielplattform auf ein plattformspezifisches Modell abgebildet wird. Dieser Transformationsprozess ist allerdings händisch durchzuführen.

Vergleicht man ECL mit MDA, dann zeigt sich, dass ECL konzeptionell eine konkrete Spezialisierung von MDA darstellt. Statt MOF wird in ECL die Architekturbeschreibungssprache Acme eingesetzt. Sie stellt damit in dem bei MOF verwendeten Sprachgebrauch ein Metametamodell dar. Die Unterstützung spezifischer Zielplattformen, die in MDA hauptsächlich durch UML-Profile bereitgestellt wird, geschieht in ECL mittels Vokabularen, die so als Metamodelle fungieren. Eine Applikationsarchitektur ist ein Modell, welches dann übersetzt und gestartet wird. In MDA entspricht das dem finalen plattformspezifischen Modell. Die laufende Applikation ist schließlich das konkrete System.

Im Gegensatz zu MDA spielen Transformationen in ECL nur eine geringe Rolle, da für die Erstellung der Applikationsarchitektur das Basismodell nur angereichert wird. Nur bei dem letzten Schritt, der Übersetzung, findet eine komplexe Transformation statt.

Es ist zu dem aktuellen Zeitpunkt noch unklar, welche Einschränkungen sich bei ECL durch die Wahl von Acme als Basismodell im Vergleich zu dem wesentlich allgemeineren MOF ergeben. Um diese Frage zu untersuchen, erscheint es sinnvoll, das gesamte ECL-System nochmals unter Einsatz des MDA-Ansatzes neu zu konzipieren und zu entwickeln. Viele Erfahrungswerte, die sich bei der Entwicklung von ECL ergeben haben, können dabei eingehen. An diesem neuen System ließen sich dann auch die neuen Möglichkeiten untersuchen, die sich durch Transformationen zwischen Modellen ergeben.

Dieses ist allerdings Gegenstand zukünftiger Forschung.