11. Architekturbasierte Konfiguration verteilter Informations- und Kontrollsysteme

Das in Kapitel 7 vorgestellte Modul eignet sich hauptsächlich für rechenintensive Applikationen. Die einzelnen Komponenten beschreiben die Nutzung entfernter Rechendienste und die lokale Ausführung von Java-Code. Ihre Interaktion definiert sich über das Arbeitsablauf- und das Datenflussparadigma. Die Architekturelemente eignen sich also gut, um Applikationen zu entwickeln, die eine weitgehend feste Reihenfolge einzelner Rechenoperationen besitzen.

Es gibt aber auch eine andere wichtige Klasse verteilter Applikationen, die *verteilten Informations- und Kontrollsysteme (VIKS)*, die i.A. von mehreren Benutzern gleichzeitig benutzt werden. Die einzelnen verteilten Komponenten sind potenziell aktiv, besitzen also einen eigenen Kontrollfluss. Sie bieten über spezielle Schnittstellen Dienste an, für deren Erbringung sie evtl. andere verteilte Komponenten benutzen. Sie stehen damit in starkem Gegensatz zu den passiven Rechendienstanbieter mit uniformer Schnittstelle aus Kapitel 7.

Eine derartige Applikation kann auch Komponenten benutzen, die nicht von ihr selber verwaltet werden, wie z.B. Informationsdienste anderer Anbieter. Ihre Struktur ist hoch dynamisch, da sie sowohl von der aktuellen Menge der Benutzer als auch von den aktiven Komponenten selbst abhängt.

Abbildung 11.1 stellt eine Überwachungsapplikation als Beispiel für ein VIKS schematisch dar. Die Darstellung lehnt sich dabei sehr frei an UML-Instanzdiagramme an. Das System enthält zwei Sensoren, eine Kamera und einen Fenstersensor. Diese werden von je einem Objekt mit einer fernaufrufbaren Schnittstelle verwaltet. Beide Objekte müssen stets dort sein, wo die eigentlichen Sensoren sich befinden. Sie sind damit bzgl. ihres Ortes beschränkt.

Die von der Kamera erzeugten Bilder werden automatisch hinsichtlich eines unbefugten Eindringens ausgewertet. Das Objekt, welches den Zugriff auf die Kamera verwaltet bietet hierfür eine Schnittstelle an, die in dem Diagramm durch einen gefüllten Kreis dargestellt ist. Den Zugriff auf diese Schnittstelle von dem Bildauswertungsobjekt wird durch einen ungefüllten Kreis dargestellt, welcher mit der Schnittstelle verbunden ist. Um das lokale Netzwerk nicht mit dem Datentransport zwischen Kamera und der Bildauswertung zu belasten, sollen beide Objekte an dem gleichen Ort liegen. Dies wird durch die Annotation der Verbindung mit der Beschränkung `kolokal` ausgedrückt.

Das zentrale Objekt in diesem Beispiel ist die Steuerung, welche die Ergebnisse der Bildauswertung und des Fenstersensors verwaltet. Diese kann

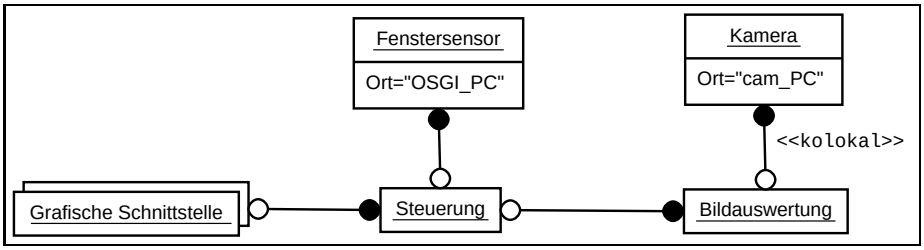

Abbildung 11.1.: Ein motivierendes Beispiel eines verteilten Informations- und Kontrollsystems

bei Bedarf über eine grafische Schnittstelle von einer oder mehreren Personen konfiguriert und kontrolliert werden. Die Anzahl der Schnittstellenobjekte in dem System ist also nicht fest bestimmt und kann sich dynamisch ändern. Dieses ist in dem Diagramm durch das zusätzliche Rechteck hinter dem Schnittstellenobjekt angedeutet. Mit einer derartigen Annotation lässt sich ein dynamisches System mit einer statischen Architektur beschreiben.

Es lässt sich leicht erkennen, dass das für die Entwicklung rechenintensiver Applikationen vorgestellte Architekturvokabular nicht adäquat ist, um derartige Applikationen auszudrücken. Der Grund liegt darin, dass dort der genaue Ablauf der Applikation beschrieben ist, während dies bei VIKS sowohl sehr schwierig als auch nicht unbedingt erforderlich ist. Die Komponenteninteraktion bei VIKS ergibt sich hauptsächlich aus der Semantik der Komponenten selbst und nicht aufgrund einer externen Steuerung.

Das bedeutet, dass eine architekturbasierte Koordinationssprache, die Komponenten als Black-Boxes betrachtet, sich nicht als vollständige Programmiersprache von VIKS eignet. Sie eignet sich aber sehr wohl zur Beschreibung der Abhängigkeiten zwischen den einzelnen Komponenten. Eine solche Beschreibung ermöglicht es dann, eine verteilte Applikation aufzusetzen und zu terminieren, wobei auch nur Applikationsausschnitte behandelt werden können. Letzteres ist vor allem bei einer Applikation mit dynamischer Struktur sinnvoll, in der die Anzahl von Klienten zur Laufzeit variiert. Eine Spezifikation der Abhängigkeiten ermöglicht es, zur Laufzeit alle für einen Klienten benötigten Komponenten zu erzeugen und mit dem schon aktiven Teil der Applikation zu verbinden.

Die architekturelle Beschreibung der Applikation stellt damit eine Beschreibung der Konfiguration der beteiligten Komponenten dar. Dabei ist mit Konfiguration sowohl die Parametrisierung der Komponenten gemeint, als auch bei Komponentenabhängigkeiten die Propagierung der Komponentenreferenzen, also das “Verdrahten” der Komponenten.

In diesem Kapitel wird an abstraktes Erweiterungsmodul vorgestellt, das grundlegende Architekturelemente und Funktionalität bereitstellt, um eine Konfiguration eines derartigen VIKS zu beschreiben. Dabei legt sich das Modul, ähnlich wie bei dem in Kapitel 6 vorgestellten Erweiterungsmodul, nicht auf ein konkretes System fest. In Kapitel 12 wird dann ein auf dem VIKS-

Modul aufsetzendes Erweiterungsmodul präsentiert, das die Entwicklung CORBA-basierter VIKS erlaubt.

Der Ansatz des VIKS-Moduls besteht in einem Architekturvokabular, welches es ermöglicht, die einzelnen verteilten Komponenten eines VIKS und deren Abhängigkeiten zu beschreiben. In der Übersetzungsphase werden zu den einzelnen Komponenten Applikationsobjekte erzeugt, welche die relevanten Abhängigkeitsinformationen in einer einfach verarbeitbaren Repräsentation enthalten. Um eine Applikation dann konkret zu starten, wird ein verteiltes Laufzeitsystem eingesetzt, welches die Applikationsobjekte benutzt, um zu diesen die benötigten Laufzeitexemplare zu erzeugen und zu konfigurieren.

Das folgende Kapitel 11.1 beschreibt das von dem VIKS-Modul bereitgestellte Architekturvokabular. Kapitel 11.2 stellt die Analysealgorithmen zur Auflösung von Abhängigkeiten und Ortsbestimmung von Komponenten vor. Abschließend beschreibt Kapitel 11.3 das Laufzeitsystem des VIKS-Moduls.

11.1. Architekturvokabular

Abbildung 11.2 gibt einen Überblick über die Elemente des Vokabulars. Alle Komponententypen sind abstrakt. Der Basiskomponententyp ist `VIKSComponent`. Dieser erlaubt es, einen Ort `location` zu spezifizieren, an dem zur Laufzeit ein Exemplar erzeugt wird. Ist kein Ort gegeben, entscheidet das Laufzeitsystem die Platzierung. Zusätzlich ist es möglich, mit `multiplicity_LB/UB` die Anzahl aller möglichen Exemplare durch untere und obere Schranken zu beschränken.

Als untere Grenze sind nur 0 und 1 gültige Werte. Für diese Entwurfsentscheidung gibt es zwei Gründe. Zum einen lassen sich die Werte 0 als auch 1 intuitiv sofort motivieren. Der Wert 0 ist der erwartungsgemäße Standardwert, während der Wert 1 für Komponenten vergeben wird, deren Existenz für die Funktionalität der Anwendung zwingend notwendig ist. Die Notwendigkeit für größere Werte ist dagegen nicht offensichtlich. Zusätzlich erschweren sie die Platzierungs- und Konfigurationsalgorithmen, da es bei mehreren Laufzeitexemplaren, die gleichzeitig erzeugt werden, nicht klar ist, wie sich deren Abhängigkeiten zu anderen Exemplaren verhalten. Dieser Punkt wird in Kapitel 11.2 klarer werden. Zusammengefasst gilt also: $0 \leq LB \leq 1 \leq UB$.

Es gibt zwei Spezialisierungen von `VIKSComponent`: `Client` und `ServiceProvider`. Letztere bieten über Anschlüsse des Typs `Interface` Dienste an, die von ersteren über `Requirement` Anschlüsse genutzt werden. Diese Anschlüsse stellen damit Anforderungen dar, die eine Komponente an ihre Architektur stellt. Diese können sowohl zur Konstruktionszeit der zugehörigen Laufzeitexemplare erfüllt werden als auch später in der Konfigurationsphase, die nach Konstruktion aller Laufzeitexemplare durchgeführt wird. Die Art der Anforderung wird über die Attribute `construction` und `configuration` spezifiziert.

Die beiden Anschlusstypen werden über Konnektoren des Typs `Association` miteinander verbunden. In den `Associated` Rollen lässt sich ähnlich wie in statischen UML-Klassendiagrammen die Kardinalität

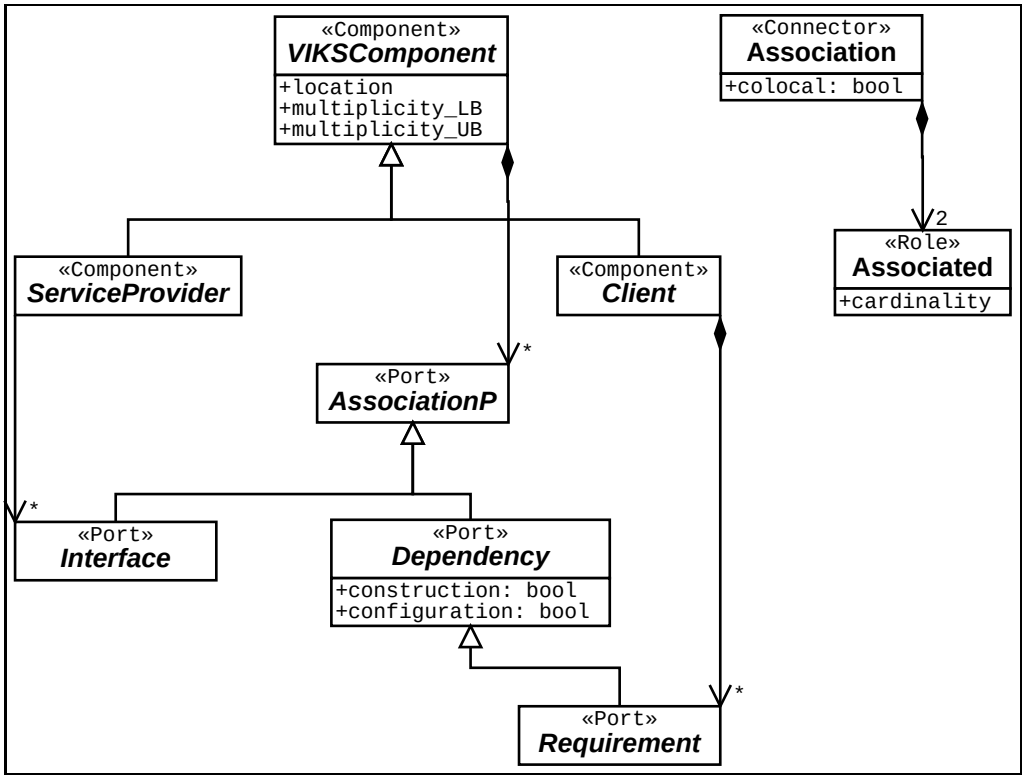

Abbildung 11.2.: Architekturvokabular für verteilte Informations- und Kontrollsysteme

der Abhängigkeit spezifizieren, die von dem Konnektor ausgedrückt wird. Zulässige Werte für das Attribut `cardinality` sind `1` und `*`. Bei einer $1 - *$ bzw. einer $* - 1$ Beziehung benutzen entweder mehrere Klienten einen Dienstanbieter oder kennt ein Klient alle Anbieter. Bei einer $1 - 1$ Beziehung gibt es immer eine eindeutige Zuordnung zwischen einem Klienten und einem Anbieter. Das bedeutet insbesondere, dass, wenn ein neuer Anbieter oder Klient erzeugt wird, automatisch das zugehörige Gegenstück erzeugt wird. Wie das konkret geschieht, wird in Kapitel 11.2 gezeigt. Eine $* - *$ Beziehung ist nicht gestattet, da hier keine eindeutige und zweckmäßige Zuordnung möglich ist. Eine Komponente, die über eine Rolle mit der Kardinalität 1 verbunden ist, muss die Vielfachheit 1 besitzen.

Wenn in einem Konnektor vom Typ `Association` das Attribut `colocal` auf `true` gesetzt ist, dann müssen ihre entsprechenden Laufzeitexemplare kolokal sein. Das bedeutet, dass sie an dem gleichen Ort erzeugt werden müssen. Sofern möglich, sollten sie sich auch im gleichen Adressraum befinden.

11.2. Laufzeitalgorithmen

Eine übersetzte VIKS-Applikation besteht aus einem Geflecht von Applikationsobjekten, welches die Abhängigkeiten der einzelnen Applikationskomponenten beschreibt. In der Übersetzung wird aus jeder VIKS-Komponente ein VIKS-Applikationsobjekt erzeugt, welches alle Informationen aus der Komponente selbst und auch zusätzliche Informationen bzgl. Abhängigkeiten zu anderen Komponenten enthält.

Zur Laufzeit wird die Applikation gestartet, indem für die einzelnen Applikationsobjekte entsprechende Laufzeitexemplare konstruiert, konfiguriert und gestartet werden. Alternativ kann auch nur ein Teil der Applikation aktiviert werden. Hierfür wählt der Anwender das Applikationsobjekt, welches er benötigt. Das Laufzeitsystem analysiert anhand der Abhängigkeiten, welche weiteren Applikationsobjekte behandelt werden müssen.

Zusätzlich zu diesem Aufsetzen ist es auch möglich ein Laufzeitexemplar E zu terminieren. Dabei werden zuerst alle Exemplare terminiert, die von E abhängen. Dieses erlaubt eine sichere Terminierung.

Um die hierfür benutzten Algorithmen beschreiben zu können, wird zuerst im Kapitel 11.2.1 ein formales Modell für die Applikationsobjekte und das Laufzeitsystem erstellt. Daraufhin werden dann die einzelnen Algorithmen vorgestellt.

11.2.1. Formale Grundlagen

Um die Algorithmen beschreiben zu können, die zur Laufzeit Verwendung finden, müssen zuerst die Applikationsobjekte und ihr Verhältnis zur ursprünglichen Applikationsarchitektur spezifiziert werden. Zusätzlich muss auch das Laufzeitsystem modelliert werden. Im folgenden werden Strukturen durch ein Tupel mit benannten Einträgen und ihrer Definitionsmenge eingeführt. Die Namen der Tupeleinträge dienen dann als Selektoren in den Al-

gorithmen. Sei z.B. die Menge X definiert als:

$$X = \mathbb{N}^2; \;\; x \in X : (a, b)$$

Dann bezeichnet für ein Element $\tilde{x} \in X$ der Ausdruck $\tilde{x}.a$ den linken Eintrag in dem Wertepaar.

Eine Applikationsarchitektur besteht, wie auch schon in Kapitel 6.1 beschrieben wurde, aus einem System aus Komponenten und Konnektoren, die über ihre Interkonnektionspunkte, den Anschlüssen (Ports) und Rollen (Roles) miteinander verbunden sind:

$$System = \text{Component} \times \text{Connectors} \times 2^{\text{Port} \times \text{Role}}$$
$$sys \in System : (components, connectors, attachments)$$

Die Relation $\underset{sys}{\longrightarrow}$ besteht aus den Paaren von Anschlüssen und Rollen, die in dem System miteinander verbunden sind.

$$p \underset{sys}{\longrightarrow} r \Leftrightarrow (p, r) \in sys.\text{attachments}$$

Ein Applikationskompilat app ist im folgenden eine Menge von Applikationsobjekten. Sei $\mathbb{A}$ die Menge aller Applikationsobjekte, dann ist $app \in 2^{\mathbb{A}}$. Im folgenden wird davon ausgegangen, dass app als globale Variable das aktuelle Applikationskompilat darstellt und sys ebenfalls als globale Variable die Applikationsarchitektur bezeichnet, aus der app durch Übersetzung entstanden ist.

Ein VIKS-Applikationsobjekt enthält nun alle Informationen über Komponenten der Applikationsarchitektur, die für die Laufzeitanalyse relevant sind:

$$\mathbb{A} = \text{Component} \times (Id_{\mathbb{L}} \cup \{\epsilon\}) \times 2^{Ref(\mathbb{A})} \times 2^{Ref(\mathbb{A})} \times 2^{Ref(\mathbb{A})} \times 2^{Ref(\mathbb{A})}$$
$$ao \in \mathbb{A} : (cp, loc, constr, conf, coloc, oTo)$$

$ao.cp$ ist die Komponente aus der Applikationsarchitektur, aus welcher das Applikationsobjekt übersetzt wurde. Wenn der Applikationsentwickler in der entsprechenden Komponente das Attribut `location` gesetzt hat, dann ist $ao.loc$ der Name des Orts in der Laufzeitumgebung, der in dem Attribut gesetzt wurde. Wurde er nicht gesetzt, ist $ao.loc$ gleich ϵ. Die Modellierung des Laufzeitsystems wird weiter unten beschrieben.

Die Menge von Referenzen auf alle Applikationsobjekte, von denen Laufzeitexemplare benötigt werden, um ein Laufzeitexemplar für dieses Applikationsobjekt ao erzeugen zu können, sei $ao.constr$. Sie ergibt sich durch Aufzählung aller Komponenten, die über einen Anschluss p_1 des Typs *Dependency* mit ao verbunden sind, wobei in p_1 das boolsche Attribut *construction* gesetzt sein muss. Die Menge $ao.constr$ ergibt sich damit wie folgt aus der Applikationsarchitektur sys und dem Kompilat app:

$$ao.constr = \&\{ao' \in app \mid (\exists p_1 \in ao.cp.\text{ports} \cap \text{Dependency})(\exists p_2 \in ao'.cp.\text{ports}) (p_1 \underset{sys}{\xrightarrow{\text{Association}}} p_2 \wedge p_1.\text{construction})\}$$

Der Operator & erzeugt eine Menge von Referenzen für alle Applikationsobjekte in der Menge, die als Parameter übergeben wurden. Er kann auch auf einzelne Applikationsobjekte angewendet werden und erzeugt dann entsprechende einzelne Referenzen.

In der obigem Definition und auch im folgenden wird ein Typ, wie z.B. *Dependency*, als Menge aller Elemente dieses Typs betrachtet. Die boolsche Funktion $\xrightarrow[sys]{T}$ über 2 Anschlüsse liefert *true* genau dann, wenn sie über einen Konnektor vom Typ T miteinander verbunden sind.

$$\begin{aligned} p_1 \xrightarrow[sys]{CnT} p_2 \iff & (\exists cn \in sys.\text{connectors} \cap CnT) \\ & (\exists r_1, r_2 \in cn.\text{roles}) \\ & (p_1 \xrightarrow[sys]{} r_1 \wedge p_2 \xrightarrow[sys]{} r_2) \end{aligned}$$

Neben den Abhängigkeiten, die schon bei der Erzeugung von Laufzeitexemplaren relevant sind, existieren noch die Abhängigkeiten, welche erst bei der Konfiguration eines Laufzeitexemplars wirken. Die Menge von Referenzen auf alle Applikationsobjekte, von denen das Applikationsobjekt *ao* ein Laufzeitexemplar zur Konfiguration benötigt, ist *ao.conf*:

$$\begin{aligned} ao.conf = \&\{ao' \in app \mid & \\ & (\exists p_1 \in ao.cp.\text{ports} \cap \text{Dependency})(\exists p_2 \in ao'.cp.\text{ports}) \\ & (p_1 \xrightarrow[sys]{\text{Association}} p_2 \wedge p_1.\text{configuration})\} \end{aligned}$$

Eine Applikationsentwickler kann fordern, dass Komponenten kolokal sind. Die Menge *ao.coloc* enthält für ein Applikationsobjekt *ao* die Menge von Referenzen auf alle Applikationsobjekte, von denen ein geeignetes Laufzeitexemplar an dem gleichen Ort wie *ao* existieren muss. Wie sich leicht erkennen lässt, ist die Kolokalität symmetrisch.

$$ao.coloc = \&\{ao' \in app \mid ao \xrightarrow{coloc} ao'\}$$

Die Kolokalitätsrelation $\xrightarrow{coloc}$ zwischen zwei Applikationsobjekten ist die transitive symmetrische Hülle über die Relation $\xrightarrow{coloc'}$, die sich aus dem Verbinden der Applikationsobjekte über einen `Association`-Konnektor mit gesetztem `colocal`-Attribut ergibt:

$$\begin{aligned} (\exists ao_1, ao_2 \in app)(& ao_1.cp \in sys.\text{components} \cap \text{VIKSComponent} \\ & ao_2.cp \in sys.\text{components} \cap \text{VIKSComponent} \\ & (\exists p_1 \in ao_1.cp.\text{ports} \cap \text{AssociationP}) \\ & (\exists p_2 \in ao_2.cp.\text{ports} \cap \text{AssociationP}) \\ & (\exists cn \in sys.\text{connectors} \cap \text{Association} \mid cn.\text{colocal} = \text{true}) \\ & (\exists r_1, r_2 \in cn.\text{roles} \mid r_1 \neq r_2) \\ & (p_1 \xrightarrow[sys]{} r_1 \wedge p_2 \xrightarrow[sys]{} r_2)) \\ \Rightarrow ao_1 \xrightarrow{coloc'} & ao_2 \end{aligned}$$

Um Applikationen mit einer dynamischen Struktur in einer statischen Architektur zu beschreiben, ist es notwendig, für ein Applikationsobjekt mehrere Laufzeitexemplare erzeugen zu können. Die Beziehungen zwischen den Applikationsobjekten übertragen sich dann auf die Laufzeitexemplare, die in der gleichen Sitzung erzeugt wurden, sofern die Applikationsobjekte in einer 1-1 Beziehung stehen. In der Applikationsarchitektur drückt sich diese Abhängigkeit durch die Kardinalität in den Rollen des Konnektors aus, welcher die beiden Komponenten miteinander verbindet. Diese müssen beide auf 1 gesetzt sein.

$$
\begin{aligned}
ao.oTo = \&\{ao' \in app \mid\ & \\
& ao'.cp \in sys.\text{components} \cap \text{VIKSComponent} \wedge \\
& (\exists p \in ao.cp.\text{ports} \cap \text{AssociationP}) \\
& (\exists p' \in ao'.cp.\text{ports} \cap \text{AssociationP}) \\
& (\exists cn \in sys.\text{connectors} \cap \text{Association}) \\
& (\exists r_1, r_2 \in cn.\text{roles} \cap \text{Associated} \mid r_1 \neq r_2) \\
& ((p \underset{sys}{\longrightarrow} r_1) \wedge (p' \underset{sys}{\longrightarrow} r_2) \wedge r_1.\text{cardinality} = 1 \wedge r_2.\text{cardinality} = 1)\}
\end{aligned}
$$

Befinden sich also zwei Applikationsobjekte sowohl in einer Kolokalitäts- als auch in einer $1-1$-Beziehung, dann überträgt sich die Kolokalitätsbeziehung auf die Laufzeitexemplare, die in dem gleichen Aufsetzvorgang erzeugt wurden. Seien a und b derartige Applikationsobjekte und $\alpha_1, \beta_1, \alpha_2, \beta_2$ zugehörige Laufzeitexemplare, die in zwei Aufsetzvorgängen erzeugt wurden, dann ist nur gefordert, dass jeweils α_1 kolokal zu β_1 und α_2 kolokal zu β_2 ist.

Ein Ort besitzt einen eindeutigen Namen und enthält Applikationsobjekte, die ihm von den Analysealgorithmen zugeordnet werden, und Laufzeitexemplare $e \in \mathbb{E}$, die an dem Ort aktiv sind.

$$
\begin{aligned}
\mathbb{L} &= Id_{\mathbb{L}} \times 2^{\mathbb{A}} \times 2^{\mathbb{E}} \\
l \in \mathbb{L} &: (id, aos, ents)
\end{aligned}
$$

Ein Laufzeitexemplar enthält den Namen des Ortes, an dem das Laufzeitexemplar aktiv ist. Sie besitzt auch eine Zahl, welche die Sitzung identifiziert, in der sie erzeugt wurde. Zusätzlich enthält sie das Applikationsobjekt, aus dem sie entstanden ist, und eine Markierung, ob sie schon konfiguriert wurde und damit einsatzbereit ist. Die Kombination von Sitzungszahl und Applikationsobjekt ergibt eine eindeutige Referenz auf das Laufzeitexemplar.

$$
\begin{aligned}
\mathbb{E} &= Id_{\mathbb{L}} \times \mathbb{N} \times \mathbb{A} \times \text{bool} \\
e \in \mathbb{E} &: (loc, ses, ao, conf)
\end{aligned}
$$

Mit diesem formalen Modell der übersetzten Applikation und des Laufzeitsystems ist es nun möglich, die Algorithmen zu formulieren. Hierfür wird ein freier Pseudocode eingesetzt, der funktionale und imperative Konstrukte enthält. Neben den gewöhnlichen Operatoren und Konstrukten wird ein Substitutionsoperator $\triangleright$ eingesetzt, der auf Mengen und Tupeln mit benannten

Einträgen definiert ist. Sei X eine Menge, dann gilt:

$$X[\alpha \triangleright \beta] = \textbf{if } \alpha \in X \textbf{then } (X \backslash \{\alpha\}) \cup \{\beta\} \textbf{ else } X$$

Bei einem Tupel t mit Einträgen $t.n_i = v_i$ wird der Name des Eintrags für die Substitution benutzt. Es gilt:

$$t[n_\xi \triangleright \gamma].n_i = \begin{cases} \gamma & : \quad i = \xi \\ v_i & : \quad i \neq \xi \end{cases}$$

Zur Steigerung der Übersichtlichkeit kann zur Definition des substituierten neuen Wertes γ auf alte Tupelinhalte zurückgegriffen werden. So besteht das Tupel $t[xs \triangleright xs \cup \{\epsilon\}]$ aus allen Einträgen von t, wobei beim Eintrag xs noch ein weiteres Element ϵ hinzugefügt wurde.

Um die Algorithmen zu vereinfachen, wird davon ausgegangen, dass in dem verteilten Laufzeitsystem zu jeder Zeit höchstens ein Aufsetz- oder Terminierungsvorgang aktiv ist. Daher können Probleme der Nebenläufigkeit hier vernachlässigt werden. Zuerst wird das allgemeine Aufsetzen von Applikationen oder von Applikationsteilen beschrieben. Danach wird die Platzierungsanalyse vorgestellt, in welcher den einzelnen Applikationsobjekten ihre Orte zugewiesen werden. Abschließend wird die Konstruktion, Konfiguration und Terminierung von Laufzeitexemplaren erläutert.

11.2.2. Aufsetzen von Teilapplikationen

Das Aufsetzen einer Applikation oder eines Teiles der Applikation geschieht in vier Schritten:

1. Zuerst wird analysiert, für welche Applikationsobjekte neue Laufzeitexemplare zu erzeugen und an welchen Orten diese zu platzieren sind. Hierfür wird die Funktion *place()* benutzt, die in Kapitel 11.2.3 beschrieben wird.

2. Dann werden die einzelnen Laufzeitexemplare konstruiert, d.h. sie werden mit den notwendigen Parametern erzeugt. Hierfür wird die Funktion *construct()* eingesetzt, die in Kapitel 11.2.4 vorgestellt wird.

3. Die konstruierten Laufzeitexemplare werden dann mit der Funktion *configure()* konfiguriert, die in Abschnitt 11.2.5 erläutert wird.

4. Abschließend werden die Laufzeitexemplare gestartet.

Abbildung 11.3 zeigt den Algorithmus zum Aufsetzen. Er erhält als Parameter

- die Applikation app mit ihren Applikationsobjekten,
- die Orte $locs$, die von dem Laufzeitsystem zur Verfügung gestellt werden,
- das Applikationsobjekt ao_{root}, welches den zu startenden Applikationsteil definiert,

- einen Ort *default* als Zielort für Applikationsobjekte, die keiner Ortsbeschränkung unterliegen,
- den boolschen Wert *all*, der spezifiziert, ob die gesamte Applikation gestartet werden soll.

```
1  deploy : 2^𝔸 × 2^𝕃 × 𝔸 × 𝕃 × bool → 2^𝕃
2  funct deploy(app, locs, ao_root, default, all) ≡
3    ses : ℕ := newID();
4    if all
5      then
6           (∀ao ∈ app)
7                (locs, _) := place(ses, app, locs, ao, unmarked,
8                                   default, false, false, all);
9      else
10          (locs, _) := place(ses, app, locs, ao_root, unmarked,
11                             default, true, false, all);
12          (∀ao′ ∈ app | ao′.cp.multiplicity_LB > 0)
13               (locs, _) := place(ses, app, locs, ao′, unmarked,
14                                  default, false, false, all);
15   fi
16   while (∃l ∈ locs)(∃ao′ ∈ l.aos)
17        locs := construct(ses, locs, ao′, unmarked);
18   end
19   while (∃l ∈ locs)(∃e ∈ l.ents | ¬e.conf)
20        locs := configure(locs, e);
21   end
22   (∀l ∈ locs)(∀e ∈ l.ents) start_ext(e);
23   return locs
24 end
```

Abbildung 11.3.: Aufsetzen einer Teilapplikation

Zuerst werden alle Applikationsobjekte, von denen neue Laufzeitexemplare zu erzeugen sind, platziert. Die Platzierung von a an einen Ort l geschieht, indem a zu $l.aos$ hinzugefügt wird. Ist die gesamte Applikation zu starten, dann werden alle Applikationsobjekte in Zeile 8 behandelt. Ist nur ein Applikationsteil aufzusetzen, dann wird ab Zeile 11 nur ao_{root} und die Applikationsobjekte, von denen es abhängt, und alle Applikationsobjekte, von denen mindestens ein Laufzeitexemplar immer vorhanden sein muss, platziert.

Daraufhin werden alle platzierten Applikationsobjekte konstruiert. D.h., an allen Orten werden die zugeordneten Applikationsobjekte durch Laufzeitexemplare ersetzt. Nach der Konfigurationsphase gilt daher:

$$(\forall l \in locs)(l.aos = \emptyset)$$

Um bei der Konstruktion $1-1$ Beziehungen auflösen zu können, wird der eindeutige Bezeichner *ses* für diesen Aufsetzvorgang benötigt.

Nachdem alle Laufzeitexemplare erzeugt wurden, werden sie in Zeile 20 konfiguriert. Abschließend werden sie gestartet. Die dafür eingesetzte Funktion $start_{\text{ext}}$ hängt von der konkreten Funktionalität der Komponente ab und muss von einem anderen Erweiterungsmodul bereitgestellt werden.

In den einzelnen Phasen wird ein Rücksetzverfahren (*Backtracking*) zur Auflösung der Abhängigkeiten benutzt. In der Platzierungsanalyse und der Konstruktion wird für die Markierung eine Funktion eingesetzt, die auf den zu markierenden Elementen definiert ist und genau dann *true* liefert, wenn das Element markiert ist. Die initiale Funktion *unmarked*, in der kein Element markiert ist, ist wie folgt definiert:

$$\text{unmarked} = (\lambda x \rightarrow false)$$

Mit der Funktion *addMark* lassen sich nun neue Markierungen hinzufügen:

$$\text{addMark}(e, m) = (\lambda e' \rightarrow \ \underline{\textbf{if}}\ (e = e')\ \underline{\textbf{then}}\ \text{true}\ \underline{\textbf{else}}\ m(e'))$$

In den einzelnen Phasen des Aufsetzens der Applikation kann es zu nicht behandelbaren Fehlern kommen, die einen Abbruch des Vorgangs erfordern. Dabei müssen alle durchgeführten Änderungen am Systemzustand zurückgesetzt werden. Dieses ist aus Gründen der Übersichtlichkeit bei der Beschreibung der Algorithmen auf die Funktion *error*() reduziert worden. In der Implementierung wird allerdings ein korrektes Aufräumen durchgeführt.

11.2.3. Platzierungsanalyse

In der Platzierungsanalyse wird bestimmt von welchen Applikationsobjekten an welchen Orten neue Laufzeitexemplare erzeugt werden. Es gibt zwei Kriterien für die Platzierung:

1. Der Ort kann in dem Applikationsobjekt selbst explizit spezifiziert sein. In diesem Fall dient die Platzierungsanalyse hauptsächlich der Überprüfung, ob sich diese Anforderung erfüllen lässt. Falls dem nicht so ist, bricht der Algorithmus ab.

2. Ein Applikationsobjekt kann mit einem anderen in einer Kolokalitätsbeziehung stehen. Es muss dann gewährleistet sein, dass beide Objekte dem gleichen Ort zugeordnet werden. Evtl. schon vorgenommene Zuordnungen müssen angepasst werden. Falls dieses nicht möglich ist, bricht der Algorithmus ab.

3. Ein Applikationsobjekt kann mit keinem anderen Applikationsobjekt in einer Kolokalitätsbeziehung stecken und keinen spezifizierten Ort besitzen. In dem Fall wird das Applikationsobjekt einem Standardort zugewiesen.

Abbildung 11.4 zeigt den Platzierungsalgorithmus. Er erhält als Parameter

- eine Zahl *ses*, welche den Aufsetzvorgang beschreibt,

- die Applikation app als Menge ihrer Applikationsobjekte,
- eine Menge *locs* aller Orte, die von dem Laufzeitsystem zur Verfügung gestellt werden,
- das zu platzierende Applikationsobjekt *ao*,
- eine Funktion *marked* zur Markierung von Applikationsobjekten,
- einen ausgezeichneten Ort $default$, dem alle Applikationsobjekte zugewiesen werden, sofern es keine weiteren Abhängigkeiten gibt,
- einen boolschen Wert *isStart*, der angibt, ob *ao* das von dem Benutzer ausgewählte Startobjekt ist,
- einen boolschen Wert *oTo*, der angibt, ob *ao* mit dem vor dem Aufruf dieser Funktion behandelten Applikationsobjekt in einer Kolokalitätsbeziehung steht,
- einen boolschen Wert *all*, der genau dann gesetzt ist, wenn die gesamte Applikation aufzusetzen und damit zu platzieren ist.

Zuerst wird überprüft, ob das zu platzierende Applikationsobjekt markiert ist. Falls dem so ist, wurde es schon platziert und die Analyse bricht ab. Anderenfalls wird es markiert. Daraufhin wird in Zeile 8 überprüft, ob eine Platzierung des Applikationsobjekts und die damit verbundene Erzeugung eines neuen Laufzeitexemplars notwendig ist. Diese Notwendigkeit ist gegeben,

- wenn das Applikationsobjekt von dem Benutzer explizit gewählt wurde und damit *isStart* gesetzt ist,
- wenn *ao* sich mit dem vorherigen Applikationsobjekt in einer $1-1$ Beziehung befindet,
- wenn die gesamte Applikation zu platzieren ist, und es noch keine Zuordnungen oder Laufzeitexemplare für *ao* gibt, welche von der Funktion

$$\text{countAllInstances} : 2^{\mathbb{L}} \times \mathbb{A} \rightarrow \mathbb{N}$$

  gezählt werden.

Trifft keine der Bedingungen zu, bricht der Algorithmus ab. Falls doch, wird als erster Schritt der Platzierung in Zeile 11 überprüft, ob mit einem neuen Laufzeitexemplar die obere Grenze der Laufzeitexemplare für dieses Applikationsobjekt verletzt wird. Ist dies gegeben, bricht der Algorithmus mit einem Fehler ab.

Ab Zeile 15 wird *ao* nun entweder an dem für *ao* spezifizierten Ort $ao.loc$ oder an dem Standardort $default$ platziert. Da $ao.loc$ nur den Namen des Ortes enthält, muss der konkrete Ort mit der Hilfsfunktion *lookup()* zuerst lokalisiert werden. Nun werden rekursiv alle Applikationsobjekte platziert, die *ao* benötigt. Das sind alle Objekte, die von den Elementen der Vereinigungsmenge von $ao.constr$ und $ao.conf$ referenziert werden. Für die Dereferenzierung

```
1  place : ℕ × 2^𝔸 × 2^𝕃 × 𝔸 × (𝔸 → bool) × 𝕃 × bool × bool × bool → (2^𝕃 × (𝔸 → bool))
2  funct place(ses, app, locs, ao, marked, default, isStart, oTo, all) ≡
3    if marked(ao)
4      then return (locs, marked)
5      else marked := addMark(ao, marked);
6                                                  // Ist Platzierung notwendig?
7           if (¬(isStart ∨ oTo∨
8                 (all ∧ countAllInstances(locs, ao) = 0))
9             then return (locs, marked)
10            else                                 // Ist Komponentenbegrenzung verletzt?
11                 if (ao.cp.multiplicity_UB ≤ countAllInstances(locs, ao))
12                   then error()
13                 fi;
14                                                 //suche Platz für übergebenes Objekt
15                 if (ao.loc = ϵ)
16                   then locs := locs[default ▷ default[aos ▷ (aos ∪ {ao})];
17                    else loc_ao := lookup(locs, ao.loc);
18                         locs := locs[loc_ao ▷ loc_ao[aos ▷ (aos ∪ {ao})]];
19                 fi;
20                                                 // platziere abhängige Objekte
21                 (∀ao′ ∈ ∗ao.constr ∪ ∗ao.conf)
22                       (locs, marked) := place(ses, locs, ao′, marked, default,
23                                               false, ao′ ∈ ∗ao.oTo, all);
24                                                 // erzeuge Kolokalität
25                 es_coloc := {e | (∃l ∈ locs)(e ∈ l.ents) ∧ (e.ao ∈ ∗ao.coloc)∧
26                                  (ao ∉ ∗e.ao.oTo)}
27                 if (∃e1, e2 ∈ es_coloc)(e1.loc ≠ e2.locs) then error() fi
28                 aos_coloc := {(ao′, l′) ∈ app × locs | ao′ ∈ ∗ao.coloc ∧ ao′ ∈ l′.aos}
29                 ls_coloc := {l ∈ locs | (∃e ∈ es_coloc)(e ∈ l.ents)∨
30                                         (∃(ao′, _) ∈ ∗aos_coloc)(lookup(locs, ao′.loc) = l)};
31                 ls_coloc := ls_coloc\{ϵ};
32                 if (| ls_coloc |> 1) then error(); fi
34                 if (| ls_coloc |= 1)
35                   then l_dest := getElement(ls_coloc);
36                    else l_dest := default;
37                 fi
38                 (∀(l′, ao′) ∈ ∗aos_coloc ∪ {ao})   locs := move(locs, ao′, l′, l_dest);
40                 return (locs, marked)
41          fi
42   fi
```

Abbildung 11.4.: Platzierungsanalyse

wird der *-Operator eingesetzt, der als Umkehrfunktion zu dem &-Operator sowohl auf Mengen als auch auf einzelne Elemente definiert ist.

Nachdem nun alle neu erzeugenden Laufzeitexemplare platziert wurden, ist jetzt die Berücksichtigung der spezifizierten Kolokalitätsbeziehungen zu gewährleisten. Dieses geschieht ab Zeile 24. Zuerst wird die Menge es_{coloc} aller Laufzeitexemplare bestimmt, die zu *ao* kolokal sind. Das sind alle Exemplare für ein Applikationsobjekt, das in *ao.coloc* referenziert wird, die nicht auch in einer 1-1 Beziehung zu *ao* stehen. Diese können noch nicht als Laufzeitexemplar vorhanden sein, da diese erst in diesem Aufsetzvorgang erstellt werden. Ein Exemplar zu einem ao', welches eine 1-1 Beziehung mit *ao* hat, wird also immer zu einem anderen *ao*-Laufzeitexemplar gehören, das in einem anderen Aufsetzvorgang erzeugt wurde. Wenn es in es_{coloc} nun zwei Exemplare gibt, die an unterschiedlichen Orten liegen, kann *ao* nicht platziert werden, und der Algorithmus bricht mit einer Fehlermeldung ab.

In dem nächsten Schritt wird die Menge aos_{coloc} aller schon platzierten Applikationsobjekte bestimmt, die kolokal zu *ao* sind. Das sind alle die Applikationsobjekte deren Platzierung evtl. an die Kolokalität angepasst werden muss. Zuerst wird allerdings überprüft, ob eine Anpassung möglich ist. Dafür wird die Menge ls_{coloc} bestimmt, die aus den Orten besteht, die kolokale Laufzeitexemplare besitzen oder von kolokalen Applikationsobjekten gefordert werden. Gibt es mehr als ein Element, dann kann die Kolokalität nicht gewährt werden und der Algorithmus bricht mit einer Fehlermeldung ab.

Enthält ls_{coloc} nun ein Element, so wird dieses als Zielort genutzt, ist die Menge leer, wird der vorgegebene Ort $default$ benutzt. Abschließend werden alle schon platzierten kolokalen Applikationsobjekt dem Zielort mit der Funktion move(ls, o, l, l') zugeordnet. Diese liefert eine Menge an Orten, die im wesentlichen ls entspricht, wobei allerdings o statt l nun l' zugeordnet ist.

$$\text{move} : 2^{\mathbb{L}} \times \mathbb{A} \times \mathbb{L} \times \mathbb{L} \rightarrow 2^{\mathbb{L}}$$

Abschließend terminiert der Algorithmus.

11.2.4. Konstruktion von Laufzeitexemplaren

In der Platzierungsphase wurden jedem Ort die Applikationsobjekte zugeordnet, für die dort neue Laufzeitexemplare generiert werden. Dieses geschieht in der Konstruktionsphase. Abbildung 11.5 zeigt den Konstruktionsalgorithmus.

Der Algorithmus erhält als Parameter einen Bezeichner *ses* des Aufsetzvorgangs, die Menge *locs* aller Orte, die von dem Laufzeitsystem zur Verfügung gestellt werden, das zu konstruierende Applikationsobjekt *ao* und eine Markierungsfunktion *marked*. Zuerst wird überprüft, ob das zu konstruierende Applikationsobjekt schon markiert wurde. Falls dem so ist, bricht der Algorithmus ab. Anderenfalls werden nach einer Markierung alle Laufzeitexemplare konstruiert, die für eine Erzeugung eines Laufzeitexemplars für *ao* nötig sind. Diese werden als Parameter für die Erzeugungsfunktion der Laufzeitexemplare benötigt.

Hierfür müssen alle Applikationsobjekte ao', die in *ao.constr* referenziert werden, betrachtet werden. Steht ao' mit *ao* in einer $1-1$ Beziehung, dann

```
1  construct : ℕ × 2^𝕃 × 𝔸 × (𝔸 → bool) → (2^𝕃 × (𝔸 → bool))
2  funct construct(ses, locs, ao, marked) ≡
3    if marked(ao)
4      then return (locs, marked)
5       else marked := addMark(ao, marked);
6            para := (λx → ϵ);
7            (∀ao′ ∈ ∗ao.constr)
8            do
9               if ¬(∃e′ | e′ = lookupPara(ses, locs, ao, ao′))
10                 then (locs, marked) := construct(ses, locs, ao′, marked);
11              fi
12              if (∃e′ | e′ = lookupPara(ses, locs, ao, ao′)
13                 then para := addPara(ses, locs, para, ao, ao′);
14                  else error();
15              fi
16           od
17           e := instantiate_ext(ses, ao, para);
18           l := getElement({l ∈ locs | ao ∈ l.aos});
19           locs := locs[l ▷ l[aos ▷ aos\{ao}]];
20           locs := locs[l ▷ l[ents ▷ ents ∪ {e}]];
21           return (locs, marked)
22   fi
```

Abbildung 11.5.: Konstruktion von Komponenten

muss ein Laufzeitexemplar als Parameter benutzt werden, die in dem gleichen Aufsetzvorgang erzeugt wurde wie das Laufzeitexemplar für *ao*. Besteht keine $1-1$ Beziehung, genügt ein beliebiges Laufzeitexemplar. Die Lokalisierung eines geeigneten Laufzeitexemplars geschieht mit der Funktion *lookupPara*(), die in Abbildung 11.6 dargestellt ist. Dabei wird die Funktion *lookupEntity*() verwendet, die je nach Eingabeparametern eine leicht unterschiedliche Semantik hat. Wird sie mit der Signatur

$$\text{lookupEntity} : \mathbb{N} \times 2^{\mathbb{L}} \times \mathbb{A} \to \mathbb{E}$$

benutzt, sucht sie in einer Menge von Orten ein Laufzeitexemplar, die für ein Applikationsobjekt in einem bestimmten Aufsetzvorgang erzeugt wurde. Wird sie dagegen mit der Signatur

$$\text{lookupEntity} : 2^{\mathbb{L}} \times \mathbb{A} \to \mathbb{E}$$

eingesetzt, sucht sie nach einem Laufzeitexemplar für ein Applikationsobjekt, ohne den Vorgang zu beachten. Auf die Beschreibung der Funktion selber wird hier verzichtet.

```
1  lookupPara : ℕ × 2^𝕃 × 𝔸 × 𝔸 → 𝔼
2  funct lookupPara(ses, locs, ao, ao_arg) ≡
3    if ao_arg ∈ *ao.oTo then lookupEntity(ses, locs, ao_arg)
4                        else lookupEntity(locs, ao_arg)
5    fi
6  end
8  addPara : ℕ × 2^𝕃 × (𝔸 → 𝔼) × 𝔸 × 𝔸 → (𝔸 → 𝔼)
9  funct addPara(ses, locs, para, ao, ao_arg) ≡
10   e_arg := lookupPara(ses, locs, ao, ao_arg);
11   (λx → if x = ao_arg then e_arg else para(x) fi)
12 end
```

Abbildung 11.6.: Lokalisieren und Hinzufügen von Parametern

In dem Konstruktionsalgorithmus wird zuerst überprüft, ob schon ein geeignetes Laufzeitexemplar vorhanden ist. Wird keines gefunden, wird in Zeile 10 eines konstruiert. Besitzt die Applikationsarchitektur eine zyklische Konstruktionsabhängigkeit, kann es passieren, dass auch nach dem Aufrufen der Konstruktionsfunktion kein entsprechendes Laufzeitexemplar vorhanden ist. In dem Falle bricht der Algorithmus mit einer Fehlermeldung ab.

Alle Laufzeitexemplare, die für die eigentliche Konstruktion nötig sind, werden in der Abbildung *para* gespeichert, welche eine Zuordnung von Applikationsobjekt zu dem zugehörigen Laufzeitexemplar darstellt. Diese Zuordnung stellt eine Vereinfachung der konkreten Implementierung dar. Sie besitzt den Nachteil, dass die konkrete Konstruktionsfunktion, die ausgeführt wird, um ein Laufzeitexemplar zu erzeugen, die anderen Applikationsobjekte kennen muss. Dieses stellt aber eine Verletzung des allgemeinen Komponentenansatzes dar, welcher fordert, dass eine Komponente möglichst unabhängig von anderen Komponenten sein sollte. Daher wird in der Implementierung, der Konstruktionsfunktion eine Abbildung der Anschlüsse auf die Laufzeitexemplare

übergeben. Dieses ermöglicht es nun, die Implementierung der Konstruktoren nur mit dem Wissen über die Komponente selbst zu realisieren.

Die Konstruktion erfolgt nun mit der Funktion $instantiate_{\text{ext}}(ses, ao, para)$, die von einem anderen Erweiterungsmodul bereitgestellt werden muss. Ihre Signatur ist:

$$\text{instantiate}_{\text{ext}} : \mathbb{N} \times \mathbb{A} \times (\mathbb{A} \to \mathbb{E}) \to \mathbb{E}$$

Abschließend wird in dem Konstruktionsalgorithmus das Applikationsobjekt ao an dem Ort l, auf den es platziert wurde, durch das neu erzeugte Laufzeitexemplar e ersetzt. Nachdem die Konstruktionsphase abgeschlossen wurde, gilt damit $(\forall l \in locs)(l.aos = \{\})$.

11.2.5. Konfiguration von Laufzeitexemplaren

In der Konfigurationsphase werden nun die erzeugten Laufzeitexemplare miteinander vernetzt. Abbildung 11.7 zeigt den hierfür eingesetzten Algorithmus. Als Parameter enthält er die Menge *locs* verfügbarer Orte und das Laufzeitexemplar e, die zu konfigurieren ist.

```
1  configure : 2^𝕃 × 𝔸 → 2^𝕃
2  funct configure(locs, e) ≡
3    if e.conf
4      then locs
5      else para := (λx → ϵ);
6           (∀ao' ∈ ∗e.ao.conf)
7           do
8               e' := lookupPara(e.ses, locs, e.ao, ao');
9               locs := configure(locs, e');
10              para := addPara(e.ses, locs, para, ao, ao');
11          od
12          e_conf := configure_ext(e, para);
13          l := lookup(locs, e);
14          locs := locs[l ▷ l[ents ▷ (ents\{e}) ∪ {e_conf}]];
15          return locs;
```

Abbildung 11.7.: Konfiguration von Komponenten

Zuerst wird überprüft, ob e schon konfiguriert ist, also $e.conf$ gesetzt ist. Falls nicht, werden ähnlich wie in der Konstruktionsphase alle Exemplare e', von denen e abhängt, in einer Funktion *para* gesammelt, welche Applikationsobjekte auf ihre entsprechenden Laufzeitexemplare abbildet. Dabei werden rekursiv alle e' ebenfalls konfiguriert. Abschließend wird e durch das konfigurierte e_{conf} ersetzt.

Die Funktion $\text{configure}_{\text{ext}}(e, para)$ ist die konkrete Konfiguration für ein Laufzeitexemplar. Diese wird von einem anderen Erweiterungsmodul zur Verfügung gestellt. Die Mindestanforderung an die Funktion besteht darin, die Konfigurationsflagge $e.conf$ zu setzen. Ihre Signatur ist:

$$\text{configure}_{\text{ext}} : \mathbb{E} \times (\mathbb{A} \to \mathbb{E}) \to \mathbb{E}$$

11.2.6. Terminierung von Teilapplikationen

Für die Terminierung eines Laufzeitexemplars e werden zunächst alle Laufzeitexemplare e' terminiert, die von e abhängen. Danach wird dann e mit der von einem anderen Erweiterungsmodul bereit gestellten Funktion $terminate_{\text{ext}}$ terminiert. Abschließend wird das terminierte Laufzeitexemplar e von seinem ehemaligen Ort entfernt.

Abbildung 11.8 zeigt den entsprechenden Terminierungsalgorithmus. Er erhält als Parameter die Menge *app* aller Applikationsobjekte der Applikation, die Menge *locs* der verfügbaren Orte, das zu terminierende Laufzeitexemplar *e*, und eine Markierungsfunktion *marked*. Wenn der Anwender ein Laufzeitexemplar auswählt, das er terminieren möchte, wird der Algorithmus mit der Markierungsfunktion *unmarked()* aufgerufen.

```
1  terminate : 2^𝔸 × 2^𝕃 × 𝔼 × (𝔼 → bool) → (2^𝕃 × (𝔼 → bool))
2  funct terminate(app, locs, e, marked) ≡
3    if marked(e)
4      then return (locs, marked)
5      else
6           marked := addMark(e, marked);
7           (∀ao' ∈ app | e.ao ∈ ∗(ao'.constr ∪ ao'.conf))
8           do
9               e' := lookupPara(e.ses, locs, ao', e.ao)
10              (locs, marked) := terminate(app, locs, e', marked);
11          od
12          terminate_ext(e);
13          l := lookup(locs, e);
14          locs := locs[l ▷ l[ents ▷ ents\{e}]];
15          return (locs, marked)
16   fi
```

Abbildung 11.8.: Terminierung von Komponenten

Dieser Algorithmus stellt sicher, dass sich die gesamte Applikation immer in einem bzgl. der in der Applikationsarchitektur definierten Abhängigkeiten konsistenten Zustand befindet. Abhängigkeiten, die nicht durch entsprechende Konnektoren in der Applikationsarchitektur ausgedrückt sind, werden allerdings nicht beachtet. Wie sich in der in Kapitel 13.3 beschriebenen Fallstudie zeigt, können durch Einsatz von Alt-Software oder durch die Nutzung spezieller Fähigkeiten des Zielsystem schnell im Code versteckte Abhängigkeiten zwischen den Laufzeitexemplaren der Applikation entstehen. Hier ist der Applikationsentwickler in der Verantwortung diese Abhängigkeiten zu erkennen und sie mit entsprechenden Konnektoren in der Applikationsarchitektur darzustellen.

11.3. Laufzeitsystem

Das Laufzeitsystem wurde in Java implementiert, wobei CORBA als Middleware zum Einsatz kam. Für die Repräsentation der Applikationsobjekte und Laufzeitexemplare der obigen Algorithmen wurden in der Implementierung die Applikationsobjektklassen eingesetzt, die von den Elementübersetzern erzeugt wurde. Ihr Aufbau orientiert sich stark an den Komponententypen des Architekturvokabulars, das in Kapitel 11.1 beschrieben wurde. Die über Konnektoren ausgedrückten Beziehungen werden in Objektvariablen gespeichert.

Die Orte werden in dem Laufzeitsystem durch sog. *Boxen* repräsentiert, die über CORBA ansprechbar sind. Abbildung 11.9 gibt einen Ausschnitt der IDL-Schnittstelle eine Box wieder. Eine Box ist jeweils einer Applikation zugeordnet und hat Zugriff auf deren Applikationsobjekte. Sie hat einen eindeutigen Bezeichner und ihre IP-Adresse als Beschreibung ihres konkreten Ortes.

```
interface Box
{
  readonly attribute string name;
  readonly attribute string appName;
  readonly attribute string location;

  string constructAppOb(in string name)
    raises (BoxException);
  void configure(in ConfigCmd configuration)
    raises (BoxException);
  oneway void start(in string entId);
  boolean terminate(in string entid, in boolean force);

  ...
};
```

Abbildung 11.9.: Ausschnitt der IDL-Schnittstelle eines Ortes im Laufzeitsystem

Die Methode `constructAppOb` erhält den Namen eines Applikationsobjekts. Diese Applikationsobjekt wird dann an dem Ort konstruiert und ein entsprechendes Laufzeitexemplar erstellt. Der Rückgabewert ist ein eindeutiger Bezeichner dieses Exemplars, der als Zeichenkette kodiert ist. Über diesen Bezeichner lassen sich Exemplare konfigurieren, starten und terminieren.

Der eigentliche Aufsetzvorgang wird von einem Server des Laufzeitsystems zur Verfügung gestellt. Er erhält die übersetzte Applikation und den Namen des zu startenden Applikationsobjekt. Wird kein Name angegeben, werden alle gestartet. Dieser Server platziert nun zuerst alle benötigten Applikationsobjekte. Daraufhin startet er Boxen an allen Orten, denen ein Applikationsobjekt zugeordnet wurde, und wo noch keine Box vorhanden ist. Abschließend konstruiert, konfiguriert und startet er die Laufzeitexemplare.

Die Benutzerschnittstelle wurde mit Servlets realisiert [Mic02]. Zusätzlich

zu dem Aufsetzen von Applikationen lassen sich auch gezielt Laufzeitexemplare terminieren. Desweiteren kann der aktuelle Status des Systems betrachtet werden. Eine genauere Beschreibung der Benutzerschnittstelle ist in Anhang A.3.2 zu finden. Für eine detailliertere Beschreibung der Implementierung sei auf [Sch02] verwiesen.