ACPLT-Technologiekonzept

Übersichtsbeschreibung

Dipl.-Ing. Lars Evertz

16.03.2016

Lehrstuhl für Prozessleittechnik
Prof. Dr.-Ing. Ulrich Epple
RWTH Aachen
D-52064 Aachen, Deutschland
Telefon +49 241 80 94339
Fax +49 241 80 92238
www.plt.rwth-aachen.de

Inhalt

1 Übersicht

Im Zentrum der ACPLT-Technologien steht ein Metamodell von Klassen, Instanzen und Beziehungen. Mit diesem Metamodell können dynamisch neue, teilweise aktive Instanzmodelle erzeugt und verwaltet werden. Um dies technologisch zu erreichen wurden die ACPLT-Objektverwaltung (OV) entwickelt. Sie besteht aus einer Laufzeitumgebung und dem besagten Metamodell. Dynamisch können weitere Klassenmodelle hinzugefügt und aus diesen Instanzmodelle generiert werden. Dabei stellt die Laufzeitumgebung nur grundlegende Funktionalitäten zur internen Manipulation der einzelnen Modellteile zur Verfügung. Die für den Nutzer sichtbare Funktionalität wird komplett mit aktiven Modellen umgesetzt. Dazu zählen unter anderem Kommunikationsmodelle, Funktionsbausteinsysteme, Ablaufsteuerungen, Darstellungsmodelle und Merkmalverwaltungssysteme. Alle umgesetzten Modelle können zur Laufzeit erkundet und manipuliert werden. Ist eines der vorhandenen Kommunikationsmodelle geladen, so können diese Aktionen von außerhalb des Laufzeitsystems erfolgen. **Abbildung 1** zeigt eine beispielhafte Modelllandschaft zur Laufzeit.

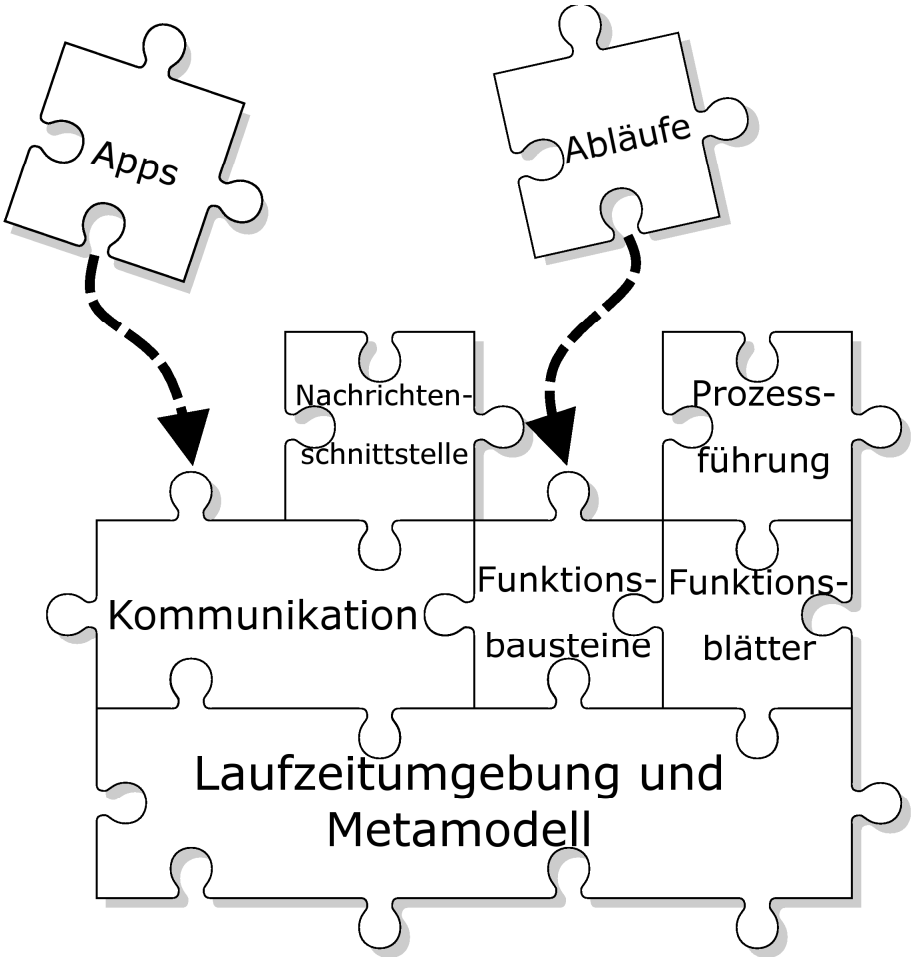

Abbildung 1: Modelllandschaft zur Laufzeit

2 Objektverwaltung

Die ACPLT-Objektverwaltung stellt zum einen ein Metamodell von Klassen, Instanzen und Verknüpfungen und zum anderen eine Laufzeitumgebung für nach dem besagten Metamodell aufgebaute dynamisch veränderliche Modelle bereit. Mit der Objektverwaltung können Instanzmodelle flexibel verwaltet werden. Dabei stehen Kommunikationsmittel für den entfernten Zugriff auf die Daten und die Struktur der Modelle zur Verfügung. Des Weiteren können zur Laufzeit neue Klassenbibliotheken hinzugefügt werden. Beim Umgang mit der Objektverwaltung werden zwei Grundideen verfolgt:

1. Die Wahrheit liegt im Zielsystem: Die Struktur und der Informationsgehalt der Daten ist im Zielsystem ohne die Notwendigkeit weiterer Informationen erkund- und veränderbar

2. Engineering (nicht Entwicklung) zur Laufzeit: Zur Laufzeit werden Instanzmodelle dynamisch erzeugt und verändert. Dabei gibt es keine Veränderungen der Programmquellen. Die notwendigen Klassen werden vorab entwickelt und können dem Objektverwaltungssystem zur Laufzeit nach Bedarf bekannt gemacht werden.

2.1 Metamodell und Klassendefinitionen

Das ov-Metamodell beschreibt das Basiswissen des Objektverwaltungssystems. Dieses Basiswissen umfasst den Aufbau und die Beziehungen von Klassen, Objekten (Instanzen), Variablen, Operationen, Kindern und Beziehungen. Das Metamodell basiert auf dem in Abbildung 2 dargestellten Metametamodell und umfasst die in Abbildung 3 dargestellten Klassen. Das Metamodell ist Bestandteil der Bibliothek *libov*.

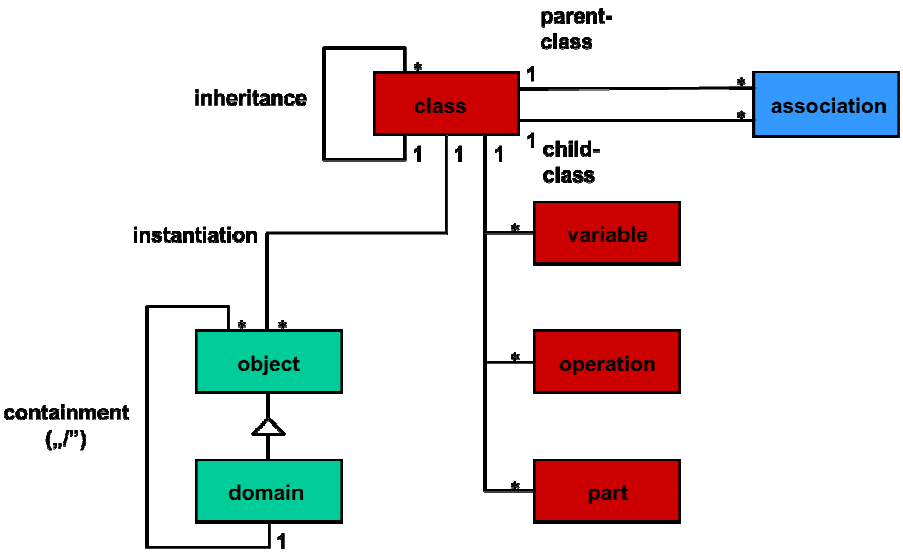

Abbildung 2: Metametamodell von ACPLT/ov

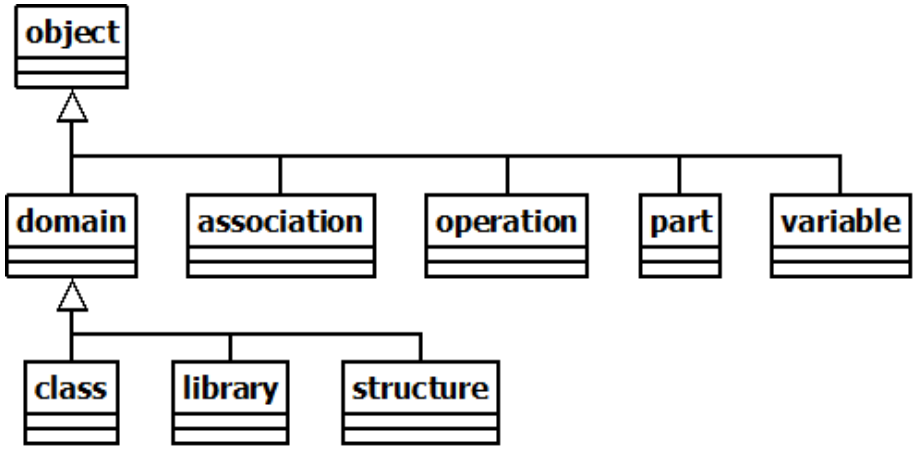

Abbildung 3: Klassen des ov-Metamodells

Das ov-Metamodell wird zur Laufzeit wie jedes andere Modell im Objektverwaltungssystem gehandhabt. Alle Klassenbeschreibungen in der Objektverwaltung bestehen also selbst aus Instanzen. Folglich ist die Beschreibung des Metamodells rekursiv aufgebaut. Basis aller Klassen ist die Klasse *object*. Instanzen von *object* tragen nur einen Identifier, ein Entstehungsdatum, einen Verweis auf ein eventuell umgebendes Objekt (siehe Abbildung 2 *containment*), ein Lebenszustandsflag und eine Tabelle von Funktionen in sich. Solche Instanzen können demnach keine Kind-Instanzen beinhalten. Diese Fähigkeit wird durch die Assoziation *containment* der Abgeleiteten Klasse *domain* (wie Abbildung 2 zeigt) ermöglicht. Abbildung 4 zeigt beispielhaft die Beschreibung der Klasse *object*, wie sie zur Laufzeit im Objektverwaltungssystem verfügbar ist.

Jede dem Objektverwaltungssystem bekannte Klasse wird durch eine Instanz der Klasse *class* beschrieben. Die Beschreibung einer Klasse, also die Klasse *class*, ist von *domain* abgeleitet und kann demnach Kind-Objekte beherbergen. Diese Kind-Objekte beschreiben Variablen, Operationen und eingebettete Instanzen, so genannte Parts. Die genannten Beschreibungsobjekte sind ihrerseits Instanzen der Klassen *variable*, *operation* und *part*. Diese Klassen sind von *object* abgeleitet.

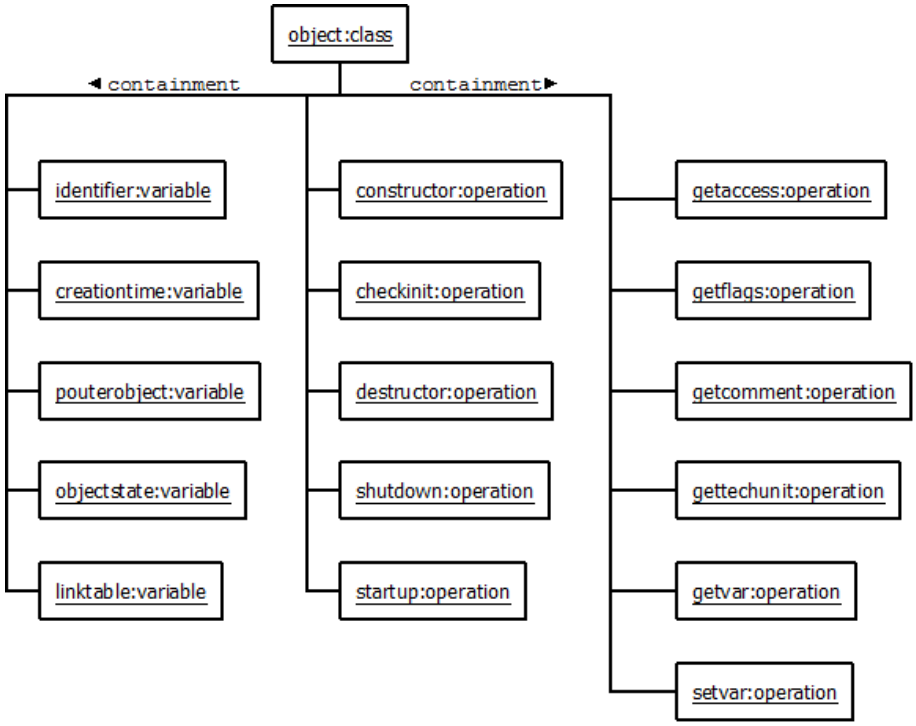

Abbildung 4: Definition der Klasse *object* im Objektverwaltungssystem

Die über die Assoziation *containment* mit der Instanz object der Klasse *class* verbundenen Instanzen beschreiben die Variablen (Instanzen der Klasse *variable*) und Operationen (Instanzen der Klasse *operation*) der definierten Klasse. Durch Hinzufügen von Instanzen der Klasse *part* können Klassen in die neu definierte Klasseeingebettet werden. Die Einbettung entspricht einer Aggregation. Die in eine Instanz eingebetteten Instanzen haben also zwingend einen gemeinsamen Lebenszyklus mit der umgebenden Instanz. Eine derartig beschriebene Klasse kann zur Laufzeit instanziiert werden. Da die Klassenbeschreibung selbst aus Instanzen besteht können folglich auch zur Laufzeit neue Klassen definiert werden. Ebenso kann die Struktur aller vorhandenen Klassen zur Laufzeit erkundet werden.

Die Klasse *object* ist oberste Basisklasse. Alle weiteren Klassen sind direkt oder indirekt von ihr abgeleitet (Assoziation *inheritance*). Daher zeigt Abbildung 4 ebenfalls die Basisinformationen, die jede Instanz in der Objektverwaltung in sich trägt und die Operationen, die jede Instanz mindestens zur Verfügung stellt. Jede Instanz trägt:

1. Einen eindeutigen Namen (identifier) im vor der übergeordneten Domäne (Assoziation *containment*) aufgespannten Namensraum;
2. Einen Zeitstempel ihres Entstehens (creationtime);
3. Einen Zeiger auf eine übergeordnete Instanz (pouterobject), falls eine Einbettung in diese als Part erfolgt ist;
4. Ein Flag (objectstate) zur Anzeige des Entstehungs- bzw. Aktivitätszustands;
5. Eine Tabelle der angelegten Verbindungen (linktable)

in sich. Außerdem stellt jede Instanz die folgenden Operationen zur Verfügung:

1. constructor: wird bei Erzeugung der Instanz ausgeführt;

2. checkinit: wird nach dem constructor ausgeführt und überprüft die Validität der Initialisierungsdaten;
3. destructor: wird bei Löschung der Instanz ausgeführt;
4. startup: versetzt das Objekt in den aktiven Zustand; wird nach der Erzeugung und bei vorhandenen Objekten bei jedem Start des umgebenden Laufzeitsystems ausgeführt;
5. shutdown: versetzt das Objekt in den inaktiven Zustand; wird vor Löschung oder vor dem Anhalten des umgebenden Laufzeitsystems ausgeführt;
6. getaccess: gibt die Zugriffsrechte für einen spezifizierten Teil des Objekts zurück;
7. getflags: gibt die Flags der zu einer Instanz gehörenden Klasse zurück;
8. getcomment: gibt den Kommentar der zu einer Instanz gehörenden Klasse zurück;
9. gettechunit: gibt die technische Einheit der zu einer Instanz gehörenden Klasse zurück;
10. getvar: gibt den Wert einer spezifizierten Variable der Instanz zurück;
11. setvar: setzt den Wert einer spezifizierten Variable der Instanz.

Diese Operationen werden von jedem Objekt bereitgestellt. Sie können je nach Klasse unterschiedlich implementiert werden. Überladungen dieser Operationen durch abgeleitete Klassen führen in der Regel nur die für sie relevanten Teile der Funktionalität aus und rufen die Funktionen der Elternklasse auf, wenn es um deren Belange geht. Die Funktion getaccess einer von *object* abgeleiteten Klasse würde demnach die Zugriffsrechte auf ihre neu definierten Variablen selbst feststellen und für die durch *object* definierten Variablen deren getaccess-Operation heranziehen.

Beziehungen zwischen Instanzen werden durch Assoziationen ausgedrückt. Die Definition einer Assoziation besteht aus einer Instanz der Klasse *association*. Die Definition dieser Klasse ist in Abbildung 5 dargestellt. Durch Instanziieren der Klasse *association,* füllen der angegebenen Variablen mit konkreten Werten und Verbinden der speziellen Verbindungen parentclass und childclass (eine weitere Rekursion in der Metamodellbeschreibung zur Laufzeit) wird eine neue Assoziation beschrieben. Verbindungen zwischen Objekten, so genannte Links, können mit den beschriebenen Assoziationen angelegt werden.

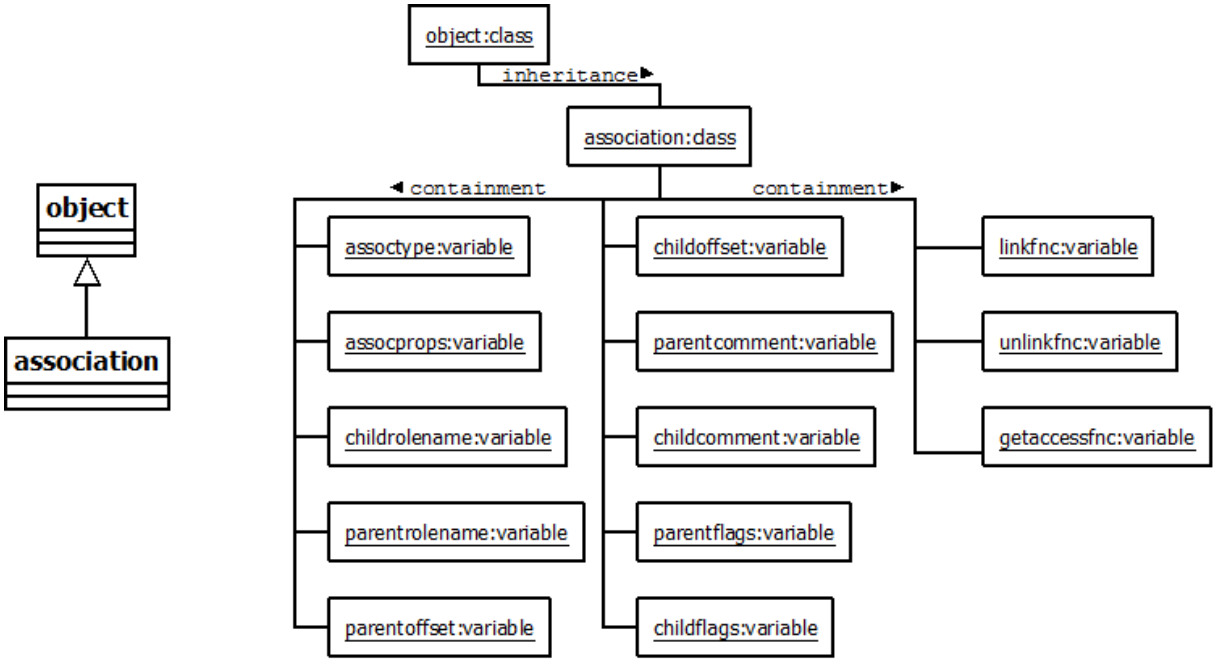

Abbildung 5: Die Klasse *association*: links ihre logische Abhängigkeit zur Klasse *object*, rechts die Definition im Ovjektverwaltungssystem.

Assoziationen werden hinsichtlich der Objekttypen, die sie verbinden können spezifiziert. Dazu hat jede Assoziationsdefinition Verbindungen zur Eltern- und zur Kindklasse, die sie verbinden kann. Die Rollennamen auf Eltern- und Kindseite sind in den Variablen childrolename und parentrolename gespeichert. Diese sind an den Intanzen abfragbar. Sie werden auch als Ankerpunkte bezeichnet.

Zur Laufzeit werden alle Verbindungen an den zugehörigen Instanzen gespeichert. Die bereits oben angesprochene Linktabelle (linktable) beinhaltet für alle Assoziationen, die für die Verbindung einer Instanz in Frage kommen, einen Eintrag. In diesen Einträgen werden die Gegeninstanzen der Verbindungen gespeichert.

2.2 Laufzeitumgebung

Die Laufzeitumgebung stellt den implementationsrahmen der Objektverwaltung dar. Sie stellt die Schnittstellen zur Interaktion mit den Modellen, eine Reihe grundlegender Funktionen, einen Scheduling-Mechanismus für aktive Modelle und die Speicherstrukturierung zur Datenhaltung bereit. Der Umgang mit den Modellen erfolgt in zwei Phasen:

1. Entwicklung: neue Klassenbibliotheken werden definiert und ausprogrammiert. Diese Phase erfolgt vor Nutzung der Modelle in der Laufzeitumgebung.

2. Engineering und Anwendung: Instanzmodelle aus den zuvor entwickelten Klassen werden im Laufzeitsystem erzeugt und verwendet.

Die Laufzeitumgebung verwendet einen dedizierten Speicherbereich, die so genannte Datenbasis oder OVD, zur Speicherung der Modelle. Dieser Speicherbereich ist nicht flüchtig und wird von der Laufzeitumgebung autonom verwaltet. In diesem Bereich sind Struktur und Daten sämtlicher Modelle, also auch der Beschreibung der vorhandenen Klassen und Klassenbibliotheken, verfügbar. Wie Abbildung 6 zeigt, werden Klassenbibliotheken (links) in Form dynamisch ladbarer Programmbibliotheken (rechts) zur Laufzeit in die Laufzeitumgebung eingebunden. Dadurch kann das Objektveraltungssystem zur Laufzeit um neue Klassenbeschreibungen erweitert werden.

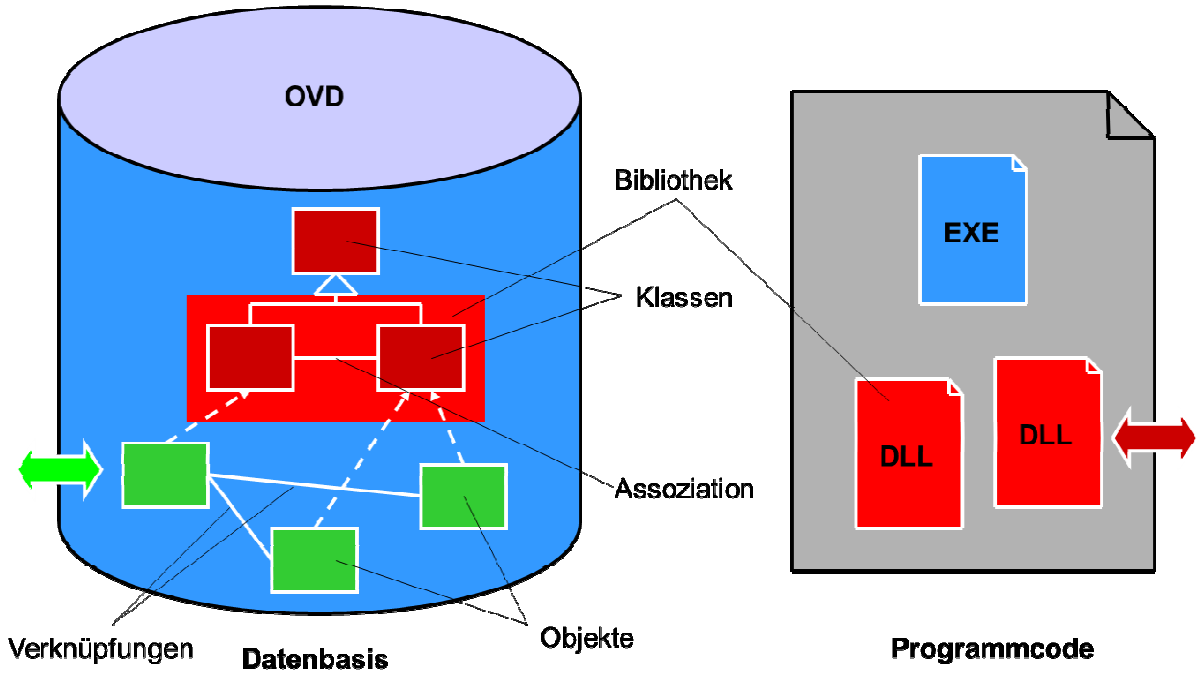

Abbildung 6: Laufzeitumgebung mit geladenen Klassenbibliotheken und Instanzmodellen links unter dem Aspekt der Datenhaltung, rechts unter dem Aspekt des Programmaufbaus

Die in den geladenen Bibliotheken definierten Klassen können nun instanziiert werden. Bei der Instanziierung wird für ein neues Objekt ein bestimmter Speicherbereich in der Datenbasis reserviert. Dann wird das neue Objekt durch Setzen seines identifiers und des Elternobjekts (Assoziation *containment*) eindeutig identifiziert. Anschließend werden die Variablen mit Initialwerten gefüllt und nacheinander constructor, checkinit und startup der Instanz ausgeführt. An diesem Punkt ist die neue Instanz aktiv und verwendbar.

Instanzen in der ov-Laufzeitumgebung sind zunächst statische, datentragende Strukturen. Durch ein Scheduling-System können zyklisch Methoden der Instanzen aufgerufen werden. Dies ermöglicht beispielsweise die Verwendung eines ov-Servers als Soft-SPS. Das Scheduling-System in der ov-Laufzeitumgebung arbeitet kooperativ und streng zyklisch. Die implementierungsbedingt schnellste Taktzeit ist eine Millisekunde. Der Ausschluss höherer Taktraten war eine Designentscheidung bei der Implementierung des Systems und ist nicht konzeptbedingt.

Theoretisch kann jede Instanz in der ov-Laufzeitumgebung ihre Methoden mit beliebigen Aufrufzyklen beim Scheduler anmelden. Bei der Verwendung als Soft-SPS sind aus Gründen des Determinismus nur zwei Einheiten im Scheduler angemeldet. Diese sind der RootComTask des Kommunikationssystems (siehe Kapitel 4) und der UrTask des Funktionsbausteinsystems. Beide Instanzen verwalten ihrerseits Listen von Kind-Tasks, die dem jeweiligen System zugerechnet werden, z. B. funktionsbausteine. Abbildung 7 zeigt die Abläufe des Taskings in einem als Soft-SPS eingesetzten ov-Server.

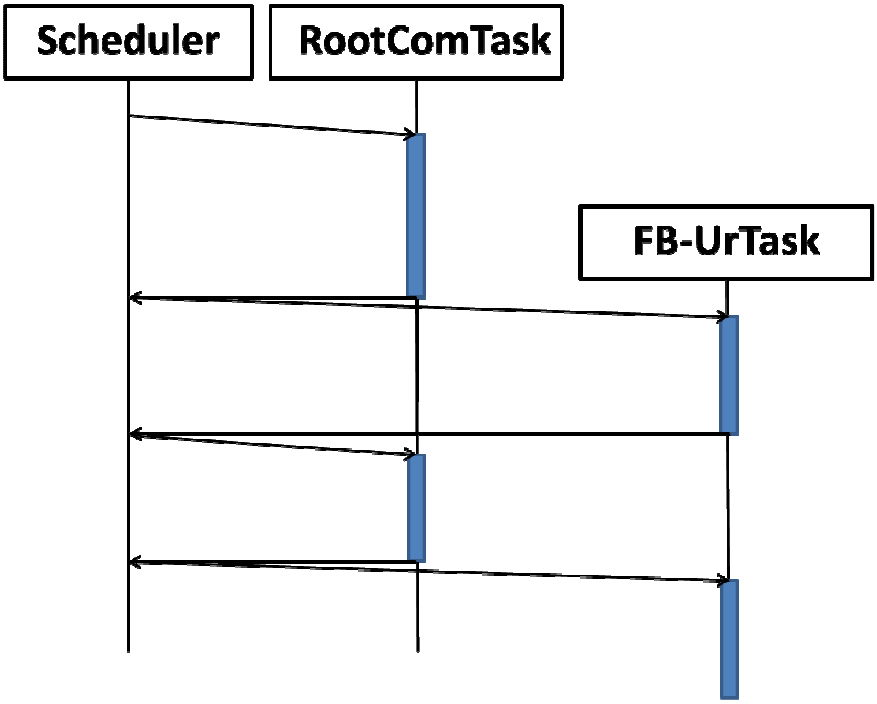

Abbildung 7: Scheduling in der ov-Laufzeitumgebung.

3 Modellhandhabung

Wie bereits erwähnt bestehen alle Modelle in einem ov-Laufzeitsystem aus Instanzen. Diese können dynamisch erzeugt, verknüpft, entknüpft und gelöscht werden. Durch die Möglichkeit neue Klassenbibliotheken zur Laufzeit einzubinden ergeben sich große Erweiterungsmöglichkeiten. Ov-Systeme können flexibel an ihre Aufgaben angepasst werden. Der grundlegende Ablauf ist dabei wie folgt:

1. Festlegung der für ein Projekt benötigten Klassen / Funktionalitäten
2. Laden der entsprechenden Klassenbibliotheken in das Laufzeitsystem
    - Gegebenenfalls Nachimplementierung einzelner Klassen
3. Erzeugen und verknüpfen der Instanzen um die gewünschte Funktionalität um zu setzen
    - Manipulation der Instanzstruktur erfolgt zur Laufzeit im Zielsystem entweder über Kommunikationsschnittstellen von außen (siehe Abschnitt 4) oder durch aktive Instanzen im Zielsystem selbst.
4. Parametrierung der Instanzen
5. Aktivierung der Instanzmethoden.

Durch umfangreiche Basisbibliotheken stehen von vorn herein vielfältige Funktionalitäten zur Verfügung. Eine Übersicht dieser Basisbibliotheken findet sich in Abschnitt 5.

4 Kommunikation

Kommunikation mit einem ov-Server kann über verschiedene Wege erfolgen. Bisher in Verwendung sind das Kommunikationssystem acplt/ks und eine Nachrichtenschnittstelle für Statuslose Kommunikation.

Das acplt/ks genannte Kommunikationssystem unterstützt die strukturelle Erkundung eines ov-Servers (Abfrage von vorhandenen Instanzen und deren Verbindungen), die Manipulation der Instanzstruktur (Erzeugen, Löschen, Verlinken, Links lösen und Umbenennen von Instanzen) sowie lesenden und schreibenden Zugriff auf die Werte von Variablen der Instanzen. Dadurch ist vollständiger Zugriff auf ein ov-Laufzeitsystem möglich.

Die Funktionalität des Kommunikationssystems basiert auf entfernten Funktionsaufrufen. Diese können entweder anhand eines nach xdr/rpc-spezifizierten Protokolls oder über eine http-Schnittstelle erfolgen. Parameter und Funktionalität sind in der ks-Protokollspezifikation festgelegt. Der Aufruf einer ks-Funktion erfolgt im Zielsystem blockierend und liefert in jedem Fall eine Antwort zurück, die mindestens über den Ausführungsstatus (Gut / Fehler) informiert.

In der aktuellen Implementierung wird die Kommunikation im ov-Laufzeitsystem von entsprechenden Instanzen abgewickelt. Das bedeutet, dass die Kommunikationsvorgänge ihrerseits im ov-Laufzeitsystem transparent verfolgbar sind. Um die Kommunikationsvorgänge responsiv zu machen, werden die Methoden der beteiligten Instanzen mit hoher Taktrate (1-5 Millisekunden) vom RootComTask aufgerufen.

Acplt/ks arbeitet auf dem bei der IANA registrierten TCP und UDP Port 7509. Unter dieser Adresse ist immer der so genannte MANAGER anzutreffen. Der MANAGER verwaltet alle ks-Server auf einem Computer. Er ordnet den benamten Servern Ports zu. Da alle ks-Kommunikation wie erwähnt durch Instanzen abgewickelt wird, kann der MANAGER ein vollwertiger ov-Server sein. Anders herum machen ov-Server nur mit einer ks-Anbindung Sinn, da die ks-Schnittstelle erst den Kontakt zur Außenwelt und damit die Manipulation zur Laufzeit ermöglicht. Die Kommunikationsanbindung als OPC-UA-Server erfolgt ebenfalls über Kommunikationsinstanzen und kann als Zusatz oder Alternative zum ks-Protokoll genutzt werden.

Neben der Funktionalität als ks-Server können ov-Systeme auch als ks-Klienten auftreten. Das bedeutet, sie können selbst andere mit ks angebundene Systeme (vorwiegend andere ov-Laufzeitsysteme) erkunden und manipulieren. Auch diese Funktionalität wird von entsprechenden Instanzen bereitgestellt. Dasselbe gilt für OPC-UA.

Der Aufbau der aktuell verwendeten ks-Bibliotheken und ihr Zusammenspiel sind im Dokument „KS-Implementierung_im_OV-Instanzraum“ beschrieben.

Neben der ks-Kommunikation wird ein statusloses Nachrichtensystem verwendet. Dieses erlaubt den Austausch von xml-basierten Nachrichten mit anderen ov-Servern oder Fremdsystemen. Der Versand von Nachrichten erfolgt dabei ohne Rückmeldung. Innerhalb eines ov-Systems werden Nachrichten durch Instanzen der Klasse *Message* dargestellt. Diese Instanzen sind wie alle anderen auch handhabbar. Versand und Empfang von Nachrichten erfolgt für den Anwender transparent durch die Instanzen des Nachrichtensystems.

Der durch Nachrichten übertragene Inhalt ist nicht weiter eingeschränkt. Daher können auf Nachrichten zum Aufruf beliebiger Funktionalitäten genutzt werden, sofern ein Empfänger vorhanden ist, der die besagte Funktionalität anbietet. Das Angebot von Funktionalität zum Aufruf durch andere Komponenten entspricht dem Dienstkonzept. Es gibt Basisbibliotheken, die dienstbasierte Interaktion mit ov-Laufzeitsystemen ermöglichen. Bei dienstbasierter Interaktion obliegt es dem anbietenden System zu entscheiden, wann es die angefragte Funktionalität ausführt. Auch wird die Funktionalität in diesem Fall weder zwingend blockierend, noch zwingend am Stück ausgeführt.

5 Basismodelle

Die Basismodelle umfassen Kommunikation (ks, Nachrichten und Dienste), Funktionsbausteine und Funktionsblätter, Ablaufbeschreibungen und Strukturbeschreibung. Auf diese Modelle wird im Folgenden näher eingegangen.

5.1 Kommunikation

Das Kommunikationsmodell ist in Anlehnung an das OSI-Schichtenmodell aufgebaut. In der untersten Ebene werden Transportbezogenen Instanzen eingesetzt, so genannte *channels*. Diese Instanzen bilden die Brücke zu verschiedenen Transport-Möglichkeiten. Bisher sind auf dieser Ebene das TCP- und das UDP-Protokoll implementiert. Es sind aber beispielsweise auch Serielle- oder Profibus-Channels denkbar. Oberhalb dieser Ebene ist die Protokollebene angeordnet. Hier werden *clientHandler* und *dataHandler* eingesetzt. Erstere werden serverseitig zur Verwaltung von Klienten eingesetzt und wirken direkt auf ihr Basissystem ein. Letztere dienen auf Klientenseite zur Verarbeitung eingehender Daten vom Server. Eine besondere Form der *dataHandler* sind *Clients*. Diese stellen definierte Methoden bereit, mit denen ks-Spezifische Aufrufe abgesetzt und die Antworten der Server verarbeitet werden können. In der Protokolleben gibt es Implementierungen des ks-Protokolls nach xdr/rpc und via http. Die Anbindung des OPC-UA-Protokolls erfolgt ebenfalls über *clientHandler*-Instanzen.

Auf Klientenseite gibt es oberhalb der Protokollebene noch eine API-Ebene. Diese wird durch die Bibliothek *ksapi* gebildet. Diese enthält Klassen, die jeweils eine spezifische Anfrage objektorientiert verfügbar machen. Es gibt beispielsweise Klassen zum Lesen und Schreiben von Variablen Werten (*getVar* und *setVar*). Die Klassen der Bibliothek *ksapi* verwenden ihrerseits die Klienten der Protokollebene.

Auf Serverseite muss der Nutzer nur die gewünschten Bibliotheken laden (*ksbase, TCPbind / UDPbind, ksxdr / kshttp / opcua*). Die Funktionalität wird dann selbstständig aufgebaut. Auf Klientenseite müssen die gewünschten Klassen vom Anwender instanziiert und parametriert werden. Um die Anwendung zu vereinfachen gibt es eine Anbindung der *ksapi* an das Funktionsbausteinsystem: die *fbcomlib*. Die Klientenfunktionalität für OPC-UA wird von der Bibliothek *opcuafb* bereitgestellt. Diese arbeitet direkt auf der Funktionsbausteinebene.

Neben den besagten funktional orientierten Kommunikationsmethoden gibt es noch den Nachrichtenversand. Dieser wird von der Bibliothek *MessageSys* umgesetzt. Ist diese Bibliothek geladen, so ist ein ov-Server in der Lage Nachrichten zu empfangen. Nachrichten werden als Objekte in einer „inbox“ genannten Domäne der Empfängerinstanz erzeugt. Das Senden von Nachrichten erfolgt durch Instanziierung und Parametrierung einer *Message*-Instanz. Der eigentliche Versand des

Objekts erfolgt durch Verlinkung mit der Instanz MessageSys vom Typ *MessageDelivery*. Beim Versand wird das ursprüngliche Nachrichtenobjekt gelöscht.

5.2 Funktionsbausteine und Funktionsblätter

Die häufigsten aktiven Instanzen in einem ov-Server sind Funktionsbausteine. Diese können in Funktionsblättern gruppiert werden. Funktionsblätter verhalten sich noch außen wie Funktionsbausteine.

5.2.1 Basissystem

Die Basisklasse der Funktionsbausteine ist die Klasse *functionblock* in der Bibliothek *fb*. Von dieser Klasse werden die funktionstragenden Klassen abgeleitet. Funktionsbausteine haben Eingangs-, Ausgangs und interne Variablen. Eingänge können von außen beeinflusst werden. Dabei wird zwischen regulären Eingängen, die mit anderen Funktionsbaustein-Variablen verbunden werden können, und Parametereingängen, die keine Verbindung erlauben, unterschieden. Interne Variablen und Ausgänge können nur vom Funktionsbaustein selbst geändert werden. Ausgänge können Quellen von Verbindungen sein. Interne Variablen können nicht verbunden werden. Lediglich das Auslesen per Kommunikationssystem ist erlaubt.

Verbindungen zwischen Variablen verbinden immer eine Quelle mit einem Ziel. Verbindungen können aktiviert und deaktiviert werden. Mehrere aktive Verbindungen können dieselbe Quelle haben. Es ist jedoch nur eine aktive Verbindung zu einem Ziel möglich. Beim Versuch weitere Verbindungen zu einem bereits verbundenen Ziel zu aktivieren werden automatisch bereits bestehende Verbindungen zu diesem Ziel deaktiviert.

Jeder Funktionsbaustein hat eine zyklisch ausgeführte Methode, die die Funktionalität bereitstellt. Die Ausführungszeitpunkte werden vom zyklisch vom ov-Scheduler aufgerufenen UrTask festgelegt. Der UrTask wird standardmäßig einmal jede Sekunde ausgeführt. Die Implementierung lässt Zykluszeiten bis hinab zu 1 ms zu, jedoch sind Zykluszeiten kleiner als 10 ms meist nicht sinnvoll, da der durch das unterlagerte Betriebssystem erzeugte Jitter dann signifikante Größen annimmt. Bei Verwendung eines Echtzeitbetriebssystems kann dieses Problem reduziert werden. Die ov-Software kann durch eine ladbare Bibliothek mit Echtzeitpriorität in einem Linux System mit rt-Preemption Patch laufen. Für andere Echtzeitbetriebssysteme ist bislang keine Anpassung erfolgt.

Jeder Funktionsbaustein wird vom UrTask angestoßen. Die Reihenfolge der Ausführung wird über die Einordnung in Tasklisten festgelegt. Die oberste Taskliste gehört zum UrTask selbst. Darunter spannt jeder Funktionsbaustein seine eigene Taskliste auf. Es werden immer zuerst die Kinder-Tasklisten ausgeführt, bevor ein Funktionsbaustein selbst ausgeführt wird. Die Zykluszeiten der einzelnen Funktionsbausteine können immer nur ein Vielfaches der Zykluszeit des UrTasks betragen. Dabei kann über die Variable iexreq festgelegt werden, ob die Ausführung eines Funktionsbausteins in jedem angegebenen Zyklus erfolgen soll (Wert TRUE) oder nur, wenn eine Eingangsvariable des Funktionsbausteins, beispielsweise über eine Verbindung oder eine Nutzereingabe, verändert wurde (Wert FALSE).

Funktionsbausteine können in Funktionsblättern (*functioncharts*) zusammengefasst werden. Diese kapseln die Funktionsbausteine strukturell und stellen eine interne Taskliste sowie Eingänge und Ausgänge zur Verfügung. Dadurch können Funktionsblätter wie Funktionsbausteine gehandhabt werden.

5.2.2 Standardfunktionsblöcke

Die in der Norm IEC61131-3 festgelegten Standardfunktionsbausteine sind in der Bibliothek *iec61131stdfb* implementiert. Sie umfassen arithmetische und logische Standardfunktionen für alle sinnvollen Datentypen. Des Weiteren sind in dieser Bibliothek einfache Zähler und Zeitglieder implementiert. Viele Funktionsbausteine unterstützen mehrere Datentypen. Dies wird durch die Verwendung von Variablen vom Typ *any* ermöglicht. Beim Umgang mit diesen Funktionsbausteinen ist zu beachten, dass Ein- und Ausgänge den gleichen Typ haben (z. B. alle SINGLE). Wird der Typ eines Eingangs geändert, so werden die anderen Eingänge und Ausgänge automatisch angepasst. Sobald eine Verbindung an einem Funktionsbaustein anliegt, können seine Eingangs- und Ausgangstypen nicht mehr verändert werden.

5.2.3 Feldanbindung

Die direkte Feldanbindung wird durch verschiedene Hardwarespezifische Bibliotheken umgesetzt. Unter anderem gibt es Möglichkeiten, Profibus-Karten, Wago-Busklemmen, Siemens WinAC Soft-SPSen und Modbus-TCP Slaves für die Feldanbindung zu verwenden. Allen genannten Wegen ist die Kodierung der jeweiligen Signaltypen gemein. Ansonsten soll hierauf nicht weiter eingegangen werden.

Oberhalb der genannten direkten Möglichkeiten wird die Bibliothek *IODriverlib* verwendet. Diese Bibliothek dient der Trennung der direkten Feldanbindung von der weiteren Logik. Die *IODriverlib* definiert Funktionsbausteine zur Umrechnung von analogen Werten und Weitergabe von digitalen Werten. Dabei können Zeitstempel der eingehenden Werte entweder übernommen (z. B. vom letzten Buszugriff) oder vom Funktionsbaustein generiert werden. Des Weiteren kann ein Simulationsmodus aktiviert werden, bei dem die Übergebenen Werte nicht mit dem Feld, sondern über andere Ein- und Ausgänge mit einer Simulation oder etwas ähnlichem verbunden werden.

5.3 Ablaufbeschreibungen

Abläufe werden in ov-Systemen mit Sequential State Charts modelliert. Dabei werden Abläufe als Folge aus Schritten und Übergangsbedingungen dargestellt. In der Implementierung im ov-System werden dazu verschiedene Klassen bereitgestellt.

Ein SSC im ov-System besitzt immer eine Instanz vom Typ *ssc*. Diese koordiniert die Abläufe und Zustandsübergänge. Schritte und Transitionen werden durch Intanzen der Klassen *Step* und *Transition* repräsentiert. Jeder Schritt besitzt drei Tasklisten entry, do und exit. Diese werden in den jeweiligen Phasen eines aktiven Schrittes aktiviert. Entry wird bei eintreten einmal getriggert, do solange der Schritt aktiv ist und exit direkt vor dem Verlassen des Schrittes. Durch Instanziierung von Aktionsrepräsentanten im containment eines Schrittes, können die jeweiligen Aktionen an den Schritt geknüpft werden. Dabei wird auch die entsprechende Phase des Schrittes festgelegt, in der die Aktionen ausgeführt werden sollen. Dies hat den Vorteil, dass mehrere Schritte dieselbe Aktion ausführen können und dabei nur der Repräsentant mehrfach erzeugt werden muss. Mögliche Aktionen sind dabei das Setzen von Variablen und das Ausführen von Funktionsbausteinen oder –blättern. Dabei können Funktionsblätter selbst SSCs beinhalten. Transitionsbedingungen werden durch Funktionsblätter implementiert. Diese Funktionsblätter müssen einen Booleschen Ausgang enthalten, die mit der entsprechenden Transition verbunden ist. Auch hier kann mehrfach dieselbe Transitionsbedingung mit verschiedenen Transitionen verknüpft werden.

Der Arbeitsablauf beim SSC-Engineering ist der folgende:

1. Festlegung des gewünschten Ablaufs
2. Instanziierung der Schritte und Transitionen
3. Verbindung der Schritte und Transitionen zur gewünschten Abfolge
4. Implementierung der Transitionsbedingungen
5. Verknüpfung der Transitionsbedingungen mit den Transitionen
6. Implementierung der Aktionen
7. Verknüpfung der Aktionen mit den entsprechenden Schritten.

Da Bedingungen und Aktionen innerhalb eines Anwendungsbereichs häufig ähnlich sind, ist es praktisch Teile von oder Ganze Bedingungen und Aktionen zu kopieren oder durch Parametrierung mehrfach verwendbar zu gestalten.

5.4 Strukturbeschreibung

Zur Beschreibung von Anlagenstrukturen können die Klassen der Bibliotheken *caex* und *Pandix* eingesetzt werden. Durch Instanziierung und Verlinkung von Repräsentationsinstanzen können Strukturen abgebildet werden. Die Instanzen können bei Bedarf um datentechnische Kopplungen an ov-Systeme erweitert werden. Damit ist es beispielsweise möglich neben der Anlagenstruktur auch den aktuellen Zustand der Anlage zu ermitteln.

Die Strukturdarstellung in Pandix kann durch eine Transformation aus einem xml-Export aus Comos erzeugt werden.

6 Erweiterungsmöglichkeiten

Ov-Laufzeitsysteme können durch interne und externe Mittel funktional erweitert werden. Intern können in erster Linie neue Klassenbibliotheken entwickelt werden. Diese können durch Dienstschnittstellen mit der Außenwelt kommunizieren oder rein lokal innerhalb des ov-Laufzeitsystems agieren. Extern können über die Dienstschnittstellen oder über die Anbindung mit ks vielfältige Funktionalitäten umgesetzt werden.

Ein wichtiges externes Werkzeug ist die Java-ks-Anbindung. Damit können in Java entwickelte Werkzeuge mit einem ov-Laufzeitsystem interagieren. Dies kann beispielsweise für regelbasiertes Engineering oder den einfachen Aufbau komplexer Strukturen aus externen Daten verwendet werden.

Durch die Dienstschnittstelle der ov-Server lässt sich ein ov-Laufzeitsystem sowohl als Anbieter, wie auch als Nutzer von Diensten verwenden. Dies wird beispielsweise in der Prozessführung oder bei der Nutzung des Merkmalsystems verwendet.

Durch diese Vielfalt der Anbindungsmöglichkeiten sind die möglichen Erweiterungen kaum begrenzt. Auch intern kann durch die Entwicklung von Nutzerbibliotheken in C (unter Verwendung beliebiger zur weiterer Bibliotheken und APIs) die Funktionalität nahezu uneingeschränkt erweitert werden.

7 Abbildungsverzeichnis