Eine Entwicklungsplattform für die architekturorientierte Programmierung

Peter Tabeling

HASSO-PLATTNER-INSTITUT für Softwaresystemtechnik
Prof. Dr. Helmert - Straße 2-3
14482 Potsdam
Peter.Tabeling@hpi.uni-potsdam.de

Zusammenfassung: Seit einigen Jahren wird die wichtige Funktion von Architekturmodellen als Grundlage der Entwicklung großer Systeme betont. Typische Aspekte dieser Modelle sind die Modellierung mit Komponenten, die Unterscheidung architektureller Sichten und Verteilungsaspekte. Obwohl die Objektorientierung diese Aspekte nicht direkt berücksichtigt, basieren gängige Entwicklungsplattformen im Wesentlichen auf objektorientierten Ansätzen. Die vorliegende Ausarbeitung stellt eine Entwicklungsplattform vor, deren Programmiermodell verschiedene architekturorientierte Konzepte umsetzt, um die genannten Aspekte besser zu berücksichtigen. Es basiert im Wesentlichen auf den Fundamental Modeling Concepts (FMC) und dem "Siemens Four Views"-Sichtenmodell.

1 Hintergrund

Es ist eine weit verbreitete Einsicht in Forschung und Praxis, dass die Erstellung großer Softwaresysteme auf Basis übergeordneter Modelle erfolgen sollte, die die "wesentlichen" Strukturen des zu erstellenden Systems erfassen und typischerweise als "Architektur" bezeichnet werden [BCK98][HNS00].

1.1 Architektur und Komponenten

Bezüglich der Frage, aus welchen Elementen die Architektur von Systemen aufgebaut ist, scheint vordergründig Konsens zu bestehen - typischerweise werden hier "Komponenten" und "Konnektoren" unterschieden. Eine derartige Sicht verbindet Architekturbeschreibungssprachen [Cl96][MT97] und wird auch in anderen Veröffentlichungen [Mo97][Sh95][BCK98] [HNS00] vertreten. Der Begriff "Komponente" wird jedoch nicht einheitlich verwendet. Für die vorliegende Arbeit sind zwei verschiedenen Deutungen zu unterscheiden:

Systemkomponenten. Im Zusammenhang mit "Softwarearchitekturen" ist unter einer Komponente ein aktiver Bestandteil des *zur Laufzeit* existierenden abstrakten[1] Systems zu

1 Der Zusatz "abstrakt" soll dabei verdeutlichen, dass mit dem Begriff "System" *nicht* eine hardwarebezogene Sicht verbunden ist.

verstehen, der eine bestimmte Funktionalität erfüllt und mit anderen Systemteilen kommuniziert. Zur Erfüllung ihrer Aufgaben verfügen diese über einen Zustand. Diese Sicht drückt sich auch in vielen Architecture Description Languages aus, die die Beschreibung eines (konzeptionellen) Systemaufbaus aus “Komponenten” und “Konnektoren” ermöglichen [Cl96][MT97]. Während unter einer “Komponente” eine primär *verarbeitende* Systemkomponente zu verstehen ist, stellt ein “Konnektor” eine primär *verbindende* Systemkomponente dar, die Kommunikation, Filterung und Pufferung von Nachrichten bietet oder auch koordinierende Aufgaben übernimmt.

Softwarekomponenten. Die obige Deutung steht jedoch im Widerspruch mit dem Komponentenbegriff der komponentenbasierten Entwicklung. Hier steht der Begriff für einen installierbaren Bestandteil der Software, der *vor der Laufzeit* des Systems relevant ist, wie z.B. eine Library. Derartige passive Artefakte können jedoch keinen Zustand im oben beschriebenen Sinne haben - eine Sichtweise, die z.B. in [Sz02] vertreten wird.

Die beiden Deutungsmöglichkeiten finden sich ansatzweise auch in der Unified Modeling Language [Ob04] wieder. Hier wird zwischen “Components” und “Artifacts” unterschieden, die sich als System- bzw. Softwarekomponenten deuten lassen.

1.2 Architekturelle Sichten

Die verschiedenen Deutungen des Komponentenbegriffs ergeben sich letztlich aus unterschiedlichen Interessenslagen bei der Betrachtung eines komplexen Softwaresystems. In [HNS00] werden dementsprechend vier unterschiedliche Sichten (“Siemens Four Views”) vorgestellt, siehe Abbildung 1.

Der “Conceptual View” erfasst dabei den anwendungsnahen, abstrakten Systemaufbau aus “Komponenten” und “Konnektoren”. Eine plattformnahe Sicht des Systems wird dagegen mit dem ”Execution View” bereitgestellt, die z.B. die Abbildung auf Kommunikationsdienste und Betriebssystemprozesse zeigt. Der “Module view” erfasst die für die Änderbarkeit wichtige Modularisierung des Programmcodes, z.B. im Sinne von Klassen und deren Vererbungsbeziehungen. Die Betrachtung von Komponenten im Sinne installierbarer, austauschbarer Softwarekomponenten erfolgt im “Code View”, bei dem “Deployment” und Konfiguration im Vordergrund stehen.

1.3 Verteilungsaspekte

Als typische Eigenschaften komplexer Systemarchitekturen sind auch Verteilung und Nebenläufigkeit zu berücksichtigen. Typische Probleme sind hier die verschiedenen Typen der “Inkonsistenz” von Daten:

Verteilungsbedingte Inkonsistenz. Aus den nicht vernachlässigbaren Verzögerungen beim Zugriff auf verteilte Daten ergibt sich eine eingeschränkte Beobachtbarkeit des globalen Systemzustandes [Mu93]. In vielen Fällen genügt es allerdings, wenn zumindest “kausale Konsistenz” [CL85] erreicht werden kann.

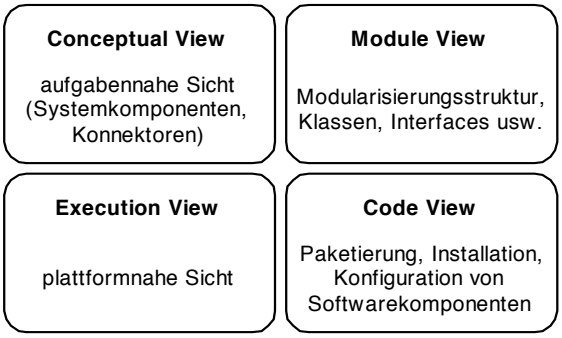


Abbildung 1: “Siemens Four Views”

Inkonsistenz gemeinsam genutzter Daten. Aufgrund unkoordinierter Zugriffe nebenläufig arbeitender Komponenten kann es bei gemeinsam genutzten Daten zu Inkonsistenzen (z.B. “dirty read”) kommen. Diese sind durch den Einsatz von Synchronisationsmechanismen wie z.B. Transaktionen [GR93] zu verhindern.

1.4 Idee der “architekturorientierten” Programmierung

Die oben diskutierten Komponententypen, architekturellen Sichten und Verteilungsaspekte stellen wichtige Merkmale komplexer Systemarchitekturen dar. Die Objektorientierung per se unterstützt diese Aspekte nicht, da z.B. Transaktionen oder Komponenten als “fremde”, zusätzliche Konzepte eingeführt werden müssen. Es erscheint daher naheliegend, diese Aspekte durch geeignete Konzepte in der Programmierung zu berücksichtigen. Im Folgenden wird ein laufendes Forschungsprojekt [Ta04b] vorgestellt, dessen Gegenstand die Entwicklung einer entsprechenden prototypischen Entwicklungsplattform ist.

2 Grundelemente des Programmiermodells

2.1 Begriffliche Basis

Das Programmiermodell basiert begrifflich auf den *Fundamental Modeling Concepts (FMC)*, einem Ansatz zur architekturorientierten Modellierung [KW03][KT02][We04] [We82][We89][Bu98]. FMC dient vor allem der verständlichen Beschreibung großer Systeme und wird bzw. wurde in verschiedenen Industrieprojekten eingesetzt [Ke03], u.a. zur Dokumentation des SAP-Systems R/3 und des Apache Web Servers [Gr01]. FMC zielt auf die Beschreibung des Conceptual View und erlaubt gleichzeitig die Beschreibung “tieferer” Systemmodelle bis hin zum Execution View, siehe links in Abbildung 2.

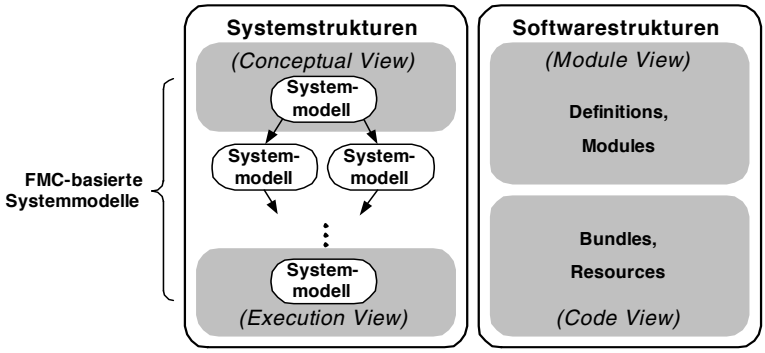


Abbildung 2: Abbildung der Grundkonzepte auf die “Siemens Four Views”

Aufgrund seiner Zielrichtung ermöglicht FMC die Beschreibung von Modellen auf verschiedenen Abstraktionsebenen sowie den Implementierungsbeziehungen zwischen Modellelementen. Ein Modell erfasst dabei die *Aufbaustruktur* aus Systemkomponenten, die darauf beobachtbaren *Ablaufstrukturen* (Verhalten) sowie die im System vorliegenden *Wertebereichsstrukturen* (Datenstrukturen).

Der in Abschnitt 1.4 skizzierten Idee folgend wurde das FMC-Metamodell um einige Elemente erweitert [Ta02a][Ta02b][Ta00a][Ta00b], um die eingangs (Abschnitte 1.1 bis 1.3) diskutierten Aspekte zu berücksichtigen:

- Abstrakte Anschlüsse sowie Annahmen bzgl. der Konsistenz verteilter Daten, siehe auch Abschnitte 2.2 bis 2.4.
- Elemente zur Unterstützung des Code bzw. Module View, siehe insbesondere Abschnitte 2.7 und 2.8.

Die verschiedenen Elemente des entstandenen Programmiermodells werden im Folgenden vorgestellt.

2.2 Systemaufbau

Akteure, Speicher und Kanäle. Zur Beschreibung des konzeptionellen Systemaufbaus unterscheidet FMC zwischen *Akteuren*, *Speichern* und *Kanälen*. Akteure sind die *aktiven* Systemkomponenten, die sämtliche Aktivitäten im System durchführen. Die Verbindung von Akteuren geschieht stets indirekt, über Kanäle oder Speicher. Diese sind rein *passiv*, d.h. sie können nur zur Ablage nicht-flüchtiger Informationen (Speicher) bzw. zur Übergabe von Nachrichten zwischen Akteuren benutzt werden (Kanäle). Speicher und Kanäle werden auch als *Orte* bezeichnet, da sie die “Stellen” im System sind, auf denen Informationen erzeugt und beobachtet werden können.

Wegen der scharfen Trennung von aktiven und passiven Komponenten ist eine Eins-zu-eins-Abbildung der beiden Kategorien auf “Komponenten” und “Konnektoren” *nicht* möglich. Beispielsweise können Konnektoren auch filternde und vermittelnde Funktionen haben, d.h. sie sind nicht immer passiv und somit nicht eindeutig einzuordnen.

Anschlüsse und Verbindungen. Akteure sollen unabhängig von der aktuellen Einbindung in den Aufbau des Systems existieren und „bausteinartig“ benutzt werden können. Dies setzt voraus, dass jeder Akteur über abstrakte *Anschlüsse* verfügt, über die er mit Speichern oder Kanälen verbunden werden kann. Eine Aufbaustruktur entsteht also erst, wenn Akteure und Orte verbunden werden, siehe Abbildung 3. (Die Pfeilrichtungen deuten Zugriffsrichtungen an, die sich aus dem Verhalten der Akteure ergeben würden.)

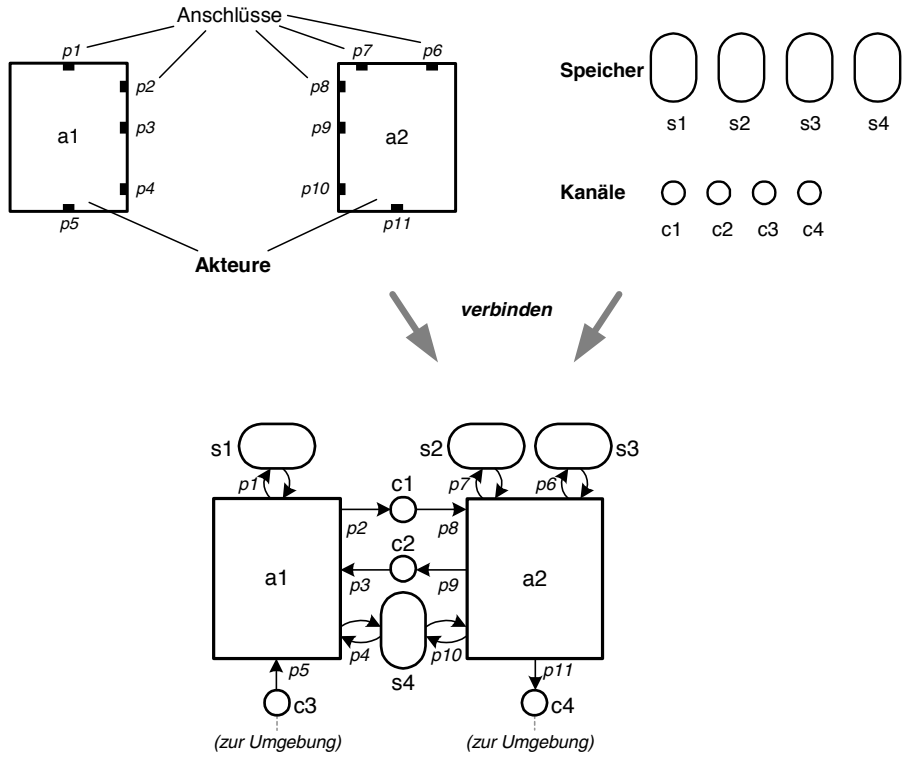


Abbildung 3: Systemaufbau aus Akteuren, Speichern und Kanälen

Akteure können mit beliebig vielen Orten verbunden werden. Da außerdem Orte mit beliebig vielen Akteuren verbunden werden können, sind selbst komplexe Aufbaustrukturen problemlos beschreibbar. Wie Abbildung 4 zeigt, können neben einfachen Kanälen und privaten Speichern auch weitergehende Konstrukte wie Broadcast-Kanäle oder gemeinsam genutzte Speicher beschrieben werden.

Grundsätzlich gilt, dass jegliche Information in explizit definierten Speichern gehalten wird. Dabei sind zwei Grundtypen zu unterscheiden:

- *Operationelle Speicher*
  Diese enthalten operationellen Zustand, aber keinen Steuerzustand. Sie werden typischerweise von mehreren Akteuren gemeinsam genutzt.
- *Threadspeicher*
  Diese enthalten neben operationellen Daten auch Steuerzustand. Da mit einem solchen Speicher eine Sequenz von Steueroperationen verbunden ist (siehe Verhaltensbeschreibung, unten), werden diese *Thread*speicher genannt. (Intern enthält ein Threadspeicher einen Stack für Steuerzustände, einen operationellen Stack sowie ggf. weitere statisch angelegte operationelle Speicher.) Typischerweise wird ein Threadspeicher nur von einem Akteur genutzt.

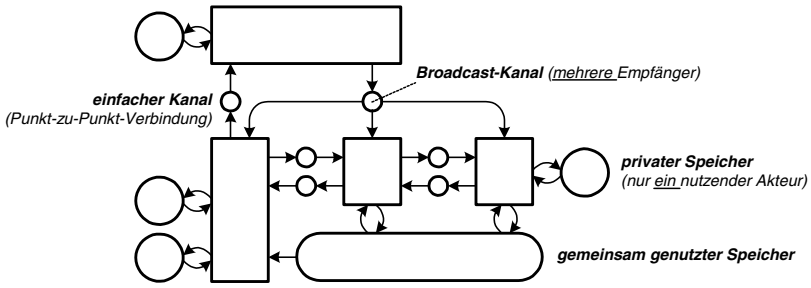
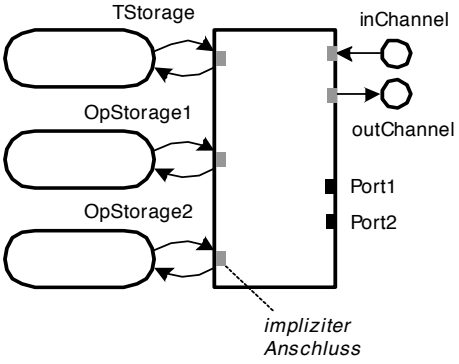


Abbildung 4: Systemaufbau - Beispiel

Bezüglich der Kanäle gilt, dass diese grundsätzlich die gleichen Inhalte wie Speicher enthalten können.

Akteure, Speicher und Kanäle eines Systemmodells können konzeptionelle Komponenten sein, die sich unmittelbar aus der Aufgabenstellung ergeben. Auf tieferen Ebenen beschreiben Akteure und Orte die Realisierung, bis hin zu plattformnahen Komponenten wie z.B. Speicher für vordefinierte Datentypen.

Abbildung 5 zeigt ein Beispiel einer Akteurstyp-Definition. Wie ersichtlich, können bei einem Akteur initial mit ihm verbundene Speicher (TStorage, OpStorage1, OpStorage2) und Kanäle (inChannel, outChannel) gegeben sein. Dies stellt nicht die indirekte Verbindung über Anschlüsse in Frage, da in diesen Fällen implizite Anschlüsse definiert werden (siehe Bild). Initial verbundene Orte sind typischerweise Speicher, über die ein Akteur wenigstens verfügen muss, um Aktivitäten ausführen zu können - dies trifft insbesondere auf Threadspeicher zu. Zu jedem Ort ist der Typ (z.B. myThreadState oder vector3d) angegeben. Wie ersichtlich ist, verfügt der Akteur über einen Threadspeicher, dem ein sequentieller Aktivitätstyp (myThreadActivity) zugeordnet ist. Außerdem sind explizit zwei (initial nicht verbundene) Anschlüsse gegeben (Port1, Port2).




```
agentType myAgentType {
   threadStorages {
      TStorage: myThreadState
         myThreadActivity
   }
   operationalStorages{
      OpStorage1: vector3d
      OpStorage2: int
   }
   channels {
      inChannel: boolean
      outChannel: int
   }
   ports {
      Port1: vector3d
      Port2: int
   }
}
```

Abbildung 5: Akteurstyp-Definition - Beispiel

2.3 Systemverhalten

Das Verhalten (Ablaufstruktur) eines Systems ergibt sich aus dem Verhalten der einzelnen Akteure. Letzteres wird allein auf Basis der Anschlüsse des jeweiligen Akteurs (bzw. den damit verbundenen Speichern und Kanälen) beschrieben, d.h. ohne direkte Bezugnahme auf andere Akteure in seiner Umgebung. (Erst dadurch wird eine “bausteinartige” Verwendung ermöglicht.)

Operationen und Zugriffe. Alle Aktivitäten von Akteuren sind auf Operationen rückführbar. Eine *Operation* ist definiert als eine Elementaraktivität, bei der ein Akteur einen Wert (das Operationsergebnis) auf einem Ort - dem Zielort - erzeugt, wobei dieser erzeugte Wert von Werten abhängt, welche der Akteur von einem oder mehreren Orten liest. Dabei findet auf jedem von einer Operation betroffenen Ort *genau ein* Zugriff statt. Ein *Zugriff* kann lesend, schreibend oder modifizierend (lesend und schreibend) sein. (Da die Zugriffe indirekt über Anschlüsse durchgeführt werden, hängt die Auswahl der betroffenen Orte auch von der Verbindungsstruktur ab.)

Durch die bewusst weit gefasste Definition des Operationsbegriffs wird eine Vielfalt praktisch relevanter Fälle abgedeckt, siehe auch die Beispiele in Abbildung 6:

- Ist ein Speicher sowohl Zielort einer Operation als auch der einzige gelesene Ort (Beispiel a), so liegt ein *Zustandsübergang* vor - mittels eines modifizierenden

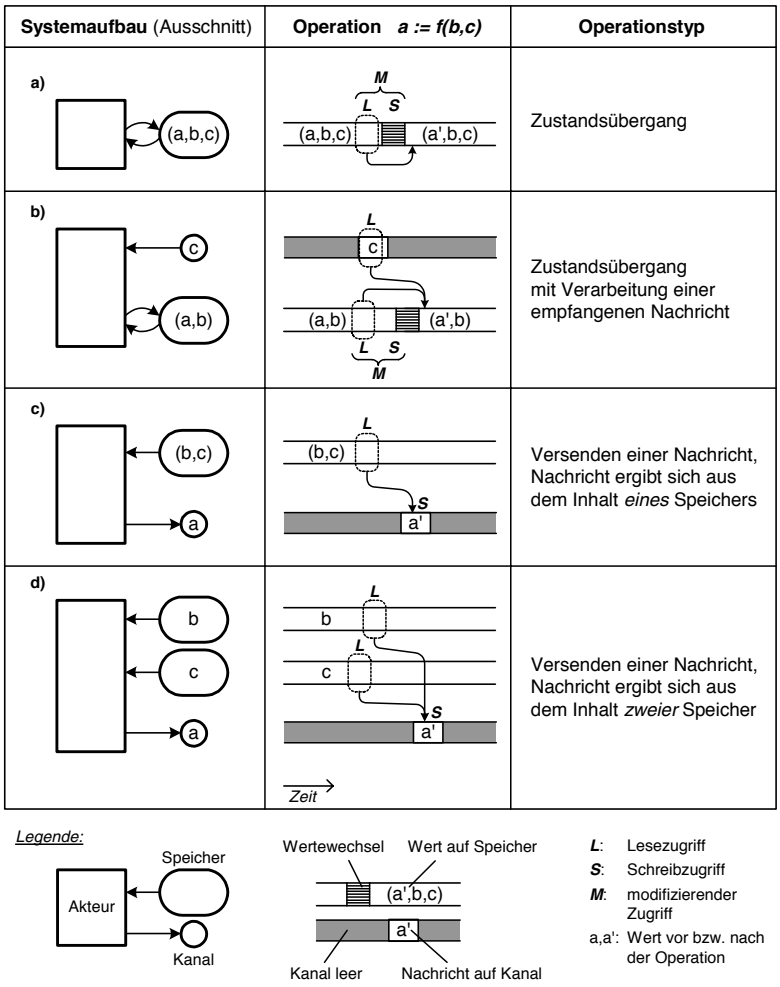


Abbildung 6: Beispiele für Operationen

Zugriffs. Ein Beispiel wäre ein lokaler Steuerzustandsübergang in einem Threadspeicher, bei dem keine Kommunikation stattfindet.

- Wird zusätzlich ein Kanal gelesen (Beispiel b), so kann auf diese Weise eine *Nachricht empfangen* werden. Hier ist ein zusätzlicher Lesezugriff auf dem Kanal gegeben.
- Ist der Zielort nicht ein Speicher, sondern ein Kanal (Beispiele c und d), dann stellt die Operation das *Versenden einer Nachricht* dar (Schreibzugriff auf den

Kanal). Da die Nachricht aus einem (Beispiel c) oder mehreren Speicherinhalten (Beispiel d) abgeleitet wird, sind dort entsprechende Lesezugriffe erforderlich.

Man beachte beim letzten Fall, dass die erzeugte Nachricht nur dann empfangen wird, wenn ein anderer Akteur auf sie wartet und sie verarbeitet (s.a. Beispiel b). Ansonsten geht die Nachricht verloren.

Die in der Entstehung befindliche Entwicklungsplattform unterstützt zunächst einfache Datentypen wie Integer, Float und String, auch als Arrays. Daraus ergeben sich die direkt unterstützten (vordefinierten) Operationstypen, wie z.B. arithmetische Operationen oder Operationen zur Stringverarbeitung.

Dynamischer Systemaufbau. Ein besonderer und wichtiger Typ von Operation ist gegeben, wenn die Aufbaustruktur des Systems selbst geändert wird, d.h. Komponenten erzeugt oder entfernt werden, oder Verbindungen zwischen diesen hergestellt oder gelöst werden. In diesem Fall gilt die Vorstellung, dass der betroffene Systemaufbau als "Inhalt" eines speziellen Speichers verfügbar ist, der als Zielort entsprechender Operationen dient. Teilstrukturen des Systemaufbaus können daher wie Datenstrukturen verarbeitet, kopiert oder gar als "Nachricht" verschickt werden.

Ereignisse und Anstoßbedingungen. Durch eine Operation (genauer: den zugehörigen Schreibzugriff) kann ein *Ereignis* verursacht werden, d.h. ein Wertewechsel in einem Zeitpunkt und auf einem Ort (dem Zielort). Dies kann das Eintreten eines neuen Zustandes sein (Speicher) oder das Erscheinen einer Nachricht (Kanal). Beide Ereignistypen können als Anstöße für weitere Operationen genutzt werden, indem entsprechende Prädikate als *Anstoßbedingungen* einer Operation definiert werden.

Verhaltenstypisierung von Akteuren - Aktivitäten. Das Verhalten eines Akteurs wird unter direkter Bezugnahme auf seine Threadspeicher festgelegt (siehe auch Abbildung 5). Im einfachsten Fall durchläuft ein Akteur eine Sequenz von Steuerzuständen. In diesem Fall muss er über genau einen Threadspeicher verfügen, in dem der Steuerzustand gehalten und in einer Folge von Threadspeicher-Operationen verändert wird. Ein Akteur kann über beliebig viele Threadspeicher - und somit auch entsprechend viele nebenläufige Steuerabläufe - verfügen.

Bei einer Threadspeicher-Operation kann eine Anstoßbedingung gegeben sein, sodass Akteure auf das Eintreffen von Nachrichten auf Kanälen reagieren können (siehe Code-Beispiel unten) oder auf bestimmte Ereignisse auf gemeinsam genutzten Speichern (Publisher-Subscriber-Szenarios).

Desweiteren können Threadspeicher-Operationen auch weitere Operationen anstoßen, z.B. das Versenden einer Nachricht über einen Kanal (siehe auch Code-Beispiel unten) oder eine Operation auf einem operationellen Speicher.

(Nicht-elementare) *Aktivitäten* sind allgemein aus mehreren Operationen aufgebaut. Die entsprechenden Anweisungen können als Sequenzen oder auch nebenläufig durchführbare Operationen (vergleichbar dem "PAR"-Konstrukt der Sprache Occam [In89]) angeordnet werden. Natürlich können Aktivitäten auch wie Unterprogramme aufgerufen werden.

Hier ein Beispiel einer Aktivitätstyp-Definition:

```
activityType myThreadActivity {

   // Aufrufparameter (hier keine):
   parameters {}

   // "lokale", auf dem operationellen Stack
   // angelegte Speicher:
   locals {
      tmp: int
   }

   // Die "eigentliche" Aktivitäts-Beschreibung:
   // Zwei Werte über inChannel empfangen, addieren und Ergebnis
   // über outChannel versenden
   procedure {
      start:  wait notEmpty(inChannel)  copy tmp,inChannel
              wait notEmpty(inChannel)  add  tmp,tmp,inChannel
                                        copy outChannel,tmp
                                        jmp  start
   }
}
```

2.4 Konsistenzannahmen beim Zugriff auf verteilte Daten

Um auch Aspekte verteilter bzw. nebenläufiger Systeme (siehe Abschnitt 1.3) zu berücksichtigen, wurden bestimmte Datenkonsistenzmodelle als integraler Bestandteil in das hier vorgestellte Programmiermodell übernommen [Ta04a]:

Der vorgestellte Operationsbegriff basiert auf der Annahme, dass beim Lesen von verschiedenen Orten individuelle, nicht vernachlässigbare Verzögerungen gegeben sind, die eine verlässliche Beobachtung des verteilten Systemzustandes verhindern. Es wird lediglich garantiert, dass die im Rahmen *einer* Operation gelesenen Werte *kausale Konsistenz* aufweisen. Die Lesezugriffe einer Operation liefern also einen (unvollständigen) "consistent snapshot" [CL85].

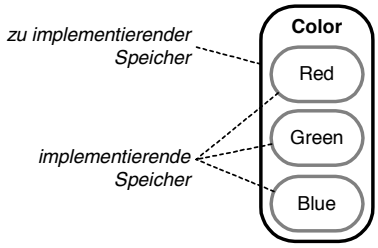
Die Möglichkeit, einen *temporal konsistenten* Wert (also einen Wert, der tatsächlich in einem Zeitpunkt vorgelegen hat) zu beobachten, bleibt auf das Lesen *einzelner* Orte beschränkt. Somit unterscheiden sich die beiden Beispiel-Operationen c) und d) aus Abbildung 6 in der Konsistenz des jeweils gelesenen Wertepaares (b,c).

Desweiteren beruht das Programmiermodell auf der Annahme, dass die einzelnen Zugriffe (auf *einen* Ort, im Rahmen *einer* Operation, siehe oben) jeweils *atomar* sind, d.h. zwei in Konflikt stehende Zugriffe werden in eindeutiger Reihenfolge durchgeführt.

2.5 Implementierungsbeziehungen

Um die Beschreibung komplexer Systeme angemessen zu unterstützen, muss es ermöglicht werden, die schrittweise Abbildung des aufgabennahen Modells (Conceptual View) auf das plattformnahe Modell (Execution View) zu beschreiben - siehe links in Abbildung 2. Dies erfordert die Festlegung von *Implementierungsbeziehungen* zwischen Modellelementen, d.h. es muss möglich sein, "höhere" Modellelemente wie Akteure, Orte und Operationen mittels "tieferer" Elemente zu beschreiben und schließlich auf das Basisrepertoire der Plattform zurückzuführen.

Implementierung von Orten und Akteuren. Ein Ort kann durch mehrere Orte implementiert werden - man denke z.B. an die Realisierung eines Speichers für Farbinformation durch mehrere "innere" Speicher für RGB-Farbkomponenten, siehe Abbildung 7:




```
locationType color {
   Red:   float
   Green: float
   Blue:  float
}
```

Abbildung 7: Ortstyp-Definition - Beispiel

Ein Akteur kann durch mehrere Akteure und evtl. zusätzliche Orte implementiert werden. Ein einfaches Beispiel dafür stellt die Realisierung eines Akteurs durch eine "innere" Aufbaustruktur aus implementierenden Akteuren und Speichern dar.

Die gerade genannten Fälle stellen typische Beispiele für *hierarchische* Verfeinerung dar. Im Gegensatz zu vielen Ansätzen ist das hier diskutierte Programmiermodell jedoch nicht auf diesen Typ beschränkt. Grundsätzlich gilt zwar, dass ein Ort bzw. Akteur durch "tiefere" Orte bzw. eine "tiefere" Aufbaustruktur implementiert wird, aber diese Abbildung kann zur Laufzeit explizit verändert werden, d.h. die Menge der einen bestimmten Akteur oder Ort implementierenden Elemente kann daher dynamisch erweitert oder reduziert werden. Dies erlaubt insbesondere die Beschreibung von Implementierungsbeziehungen, die gar nicht als hierarchische Verfeinerung modellierbar sind, wie z.B. "Pooling": Hier werden mehrere zu implementierende Akteure oder Orte implementiert, indem aus einem gegebenen Vorrat ("Pool") einzelne Akteure bzw. Orte ad hoc als Implementierung ausgewählt und später wieder freigegeben werden (Aktivierung bzw. Deaktivierung).

Implementierung von Operationen. "Höhere" Operationen sind mittels einfacherer Operationen zu implementieren. Ein einfaches Beispiel wäre die Implementierung der Operation „a:=a/b+b“ durch zwei aufeinanderfolgende Operationen „a:=a/b“ und „a:=a+b". Grundsätzlich ähnelt die Beschreibung eines Operationstyps der Beschreibung einer Aktivität (siehe Kapitel 2.3). Der wesentliche Unterschied besteht jedoch darin, dass bei einer Operation die in Kapitel 2.4 vorgestellten Konsistenzannahmen gelten, woraus

sich wichtige Unterschiede für die Behandlung bei der Programmausführung ergeben, siehe nächster Abschnitt.

2.6 Implizite Transaktionen und Snapshots

Da im Rahmen *einer* Operation für *einen* betroffenen Ort genau *ein* Zugriff durchgeführt wird, impliziert eine Implementierung "höherer" Operationen oder Orte durch mehrere "tiefere" Operationen bzw. Orte i.A. eine Abbildung eines höheren Zugriffs auf mehrere Zugriffe der tieferen Ebene. Abbildung 8 veranschaulicht dies anhand des oben erwähnten Beispiels einer Operationsfolge.

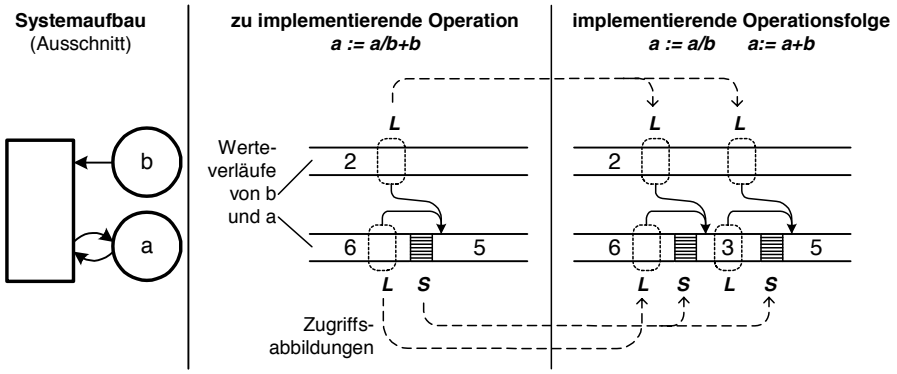


Abbildung 8: Beispiel zur Operationsimplementierung bzw. Zugriffsabbildung

Die Abbildungen von Zugriffen in Verbindung mit den in Kapitel 2.3 diskutierten Konsistenzannahmen sind von fundamentaler Bedeutung für das Programmiermodell, da sich daraus wichtige Implikationen für die Implementierung von Operationen ergeben. Diese werden hier nur kurz vorgestellt, eine ausführliche Diskussion findet sich in [Ta04a] bzw. [Ta00a].

Bezüglich der Zugriffe auf einen einzelnen Ort wurde temporal konsistente Beobachtung und Zugriffsatomarität gefordert. Daraus lassen sich für die Menge der implementierenden Zugriffe drei der so genannten "ACID-Eigenschaften" [GR93] ableiten. Atomicity, Consistency und Isolation leiten sich nämlich aus den Forderungen nach temporaler Konsistemz und Zugriffsatomarität ab, während Durability erst bei zusätzlicher Forderung nach Fehlertoleranz gegeben ist. Zugriffsmengen, die einen einzigen Zugriff auf höherer Ebene implementieren, sind daher wie *Transaktionen* durchzuführen.

Wegen der Forderung nach kausaler Konsistenz beim Lesen mehrerer Orte müssen die Lesezugriffe einer Operation auf tieferer Ebene mittels entsprechender Verfahren als "*Snapshot*" [CL85] koordiniert werden.

2.7 Modularisierungselemente

Die bislang beschriebenen Konzepte des Programmiermodells betreffen den Bereich der *System*strukturen. Im Folgenden werden diejenigen Konzepte diskutiert, die die *Software*strukturen betreffen.

Definitions. Die semantisch grundlegenden Elemente der Beschreibung sind zunächst die *Definitions*. Gegenstand jeder Definition ist die Festlegung eines nicht vordefinierten (d.h. nicht im Basisrepertoire der Sprache enthaltenen) Typs. Dabei wird zwischen Akteurstypen, Ortstypen, Aktivitätstypen, Operationstypen (siehe entsprechende Beispiele oben) und Prädikatstypen unterschieden. (Letztere dienen u.a. zur Definition neuer Anstoßbedingungen und werden hier nicht weiter betrachtet.)

Modules. Definitions sind jedoch nicht mit Modulen gleichzusetzen. Module sollen - dem Gedanken des Information Hiding folgend - möglichst eine Entwurfsentscheidung enthalten bzw. Programmteile mit starken gegenseitigen Abhängigkeiten zusammenfassen. Derartige Programmteile wären z.B. eine Ortstyp-Defintion für Speicher vom Typ “color” (siehe Beispiel in Abbildung 7) und eine Operationstyp-Definition, deren Implementierung von der Implementierung des Speichertyps “color” abhängt, z.B. ein Operationstyp “changeSaturation”, der die Farbsättigung verändert. Um solche Abhängigkeiten “lokal” zu beschränken, werden *Modules* benutzt, als Einheiten der Kapselung. Modules enthalten somit ein oder mehrere zusammenhängende Definitions. Modules können in Verwendungsbeziehung stehen (Module A verwendet die in Module B definierten Typen).

Ein Module kann wie eine Klasse in der objektorientierten Programmierung eingesetzt werden, nämlich genau dann, wenn *genau ein* Speichertyp (vgl. Objektdaten) und eine Menge davon abhängiger Operationstypen (vgl. Methoden) in einem Module zusammengefasst werden. Dies wäre allerdings ein Spezialfall, da ein Module eine beliebige Zusammenstellung von Definitions enthalten kann. Die Nähe zum Klassenkonzept legt jedoch den Gedanken nahe, Mechanismen wie Polymorphie, Kapselung und Vererbung einzuführen, da diese Konzepte auch dann erstrebenswert sind, wenn Akteure, Orte usw. an die Stelle des Objektbegriffs treten.

2.8 Paketierungskonzepte

Unter “Paketierung” wird hier die Aufteilung der zu installierenden Software auf kleinste austauschbare bzw. wiederverwendbare Installationseinheiten verstanden. Diese können einzelne oder mehrere (übersetzte) Module und sonstige Artefakte enthalten.

Resources. Eine Resource ist eine Einheit der Softwarekomposition bzw. -konfiguration, also eine kleinste, *potentiell* installierbare Software-Einheit. Eine Resource kann zwar prinzipiell isoliert von anderen Resources installiert werden, ist aber nicht notwendigerweise alleine sinnvoll nutzbar. Beispiele sind einzelne Modules, Hilfedateien, Bilder oder andere für den Betrieb einer Anwendung erforderliche Daten, einschließlich ganzer Bundles (siehe unten).

Bundles. Ein Bundle fasst Resources zusammen, die zur Nutzung/Systeminstallation nur gemeinsam installiert oder entfernt werden sollten (ähnlich den .NET-“Assemblies” und J2EE-“Archives”). Ein Bundle muss als besonderen Bestandteil auch eine selbstbezügliche Auflistung all seiner Bestandteile (Resources) haben (“Bundle Contents”). Abhängigkeiten können dadurch berücksichtigt werden, dass ein erforderliches Bundle (z.B. B) als Resource in einem darauf aufbauenden Bundle (z.B. A) enthalten sein kann. Eine Resource kann in verschiedenen Bundles enthalten sein. Derartige “shared” Resources werden nur einmal installiert und werden erst entfernt, nachdem alle darauf aufbauenden Bundles deinstalliert worden sind.

3 Kernpunkte und Implikationen des Programmiermodells

3.1 Unterstützung der architekturellen Sichten

Die vorgestellten Konzepte zielen auf die saubere Trennung der unterschiedlichen Architektursichten (siehe Kapitel 1.2) ab. Daher werden für jede dieser Sichten “zugeschnittene“ Beschreibungs- bzw. Strukturierungsmittel bereitgestellt, siehe Abbildung 2.

Entsprechend der grundlegenden Unterscheidung zwischen Software- und Systemkomponenten (siehe Abschnitt 1.1) sind Akteure, Speicher und Kanäle als reine Systemkomponenten zu betrachten, d.h. die entsprechenden Systemmodelle sind dem Conceptual View bzw. Execution View zuzuordnen. Speziell der Aufbau des “obersten” Systemmodells aus aufgabennahen Akteuren und Orten entspricht dem *Conceptual View*, wobei hier auch grundlegende Abläufe (Aktivitäten und Operationen), die wichtige Use Cases abdecken, hinzuzurechnen sind.

Das “unterste” Systemmodell, bei dem alle Typen (Akteure, Orte, Operationen, Aktivitäten) auf vordefinierte Typen der virtuellen Maschine zurückgeführt sind, entspricht dem *Execution View*. Hierzu zählt außerdem die Abbildung der Aufbaukomponenten auf die “Knoten” der (verteilten) virtuellen Maschine (siehe auch Kapitel 4.3), die sich aus der Installation ergibt.

Die weiteren Konzepte sind dem Module bzw. Code View zuzuordnen. Modules und Definitions bilden die Grundlage der Modularisierung, d.h. sie sind auf den *Module View* abbildbar. Dagegen betreffen Bundles und Resources die Aufteilung des Codes in installierbare bzw. austauschbare Einheiten - also den *Code View*.

Das Programmiermodell unterstützt somit die unterschiedlichen Komponentenbegriffe und architekturellen Sichten.

3.2 Integration von Verteilungsaspekten

Der oben diskutierte Operationsbegriff (Abschnitt 2.3) mit den daran verbundenen Konsistenzannahmen (Abschnitt 2.4) sowie die Implementierungsbeziehungen (Abschnitt 2.5) bilden integrale Bestandteile des Programmiermodells. Dies ermöglicht zur Laufzeit die

automatische Identifikation und Abwicklung der implizit gegebenen Transaktionen bzw. Snapshots (Abschnitt 2.6) durch die zur Entwicklungsumgebung gehörigen virtuellen Maschine (siehe Abschnitt 4.3). Dies bedeutet konkret, dass es nicht erforderlich ist, die Zugriffe auf verteilte bzw. gemeinsam genutzte Daten explizit mittels Sperren o.ä. Mitteln zu synchronisieren.

Die Konsistenz gemeinsam genutzter bzw. verteilter Daten wird durch das Programmiermodell implizit festgelegt.

4 Praktische Umsetzung

Das vorgestellte Programmiermodell wird in einem laufenden Forschungsprojekt umgesetzt, bei dem eine prototypische Entwicklungsplattform, genannt STAGE [Ta04b], erstellt wird. Die folgenden Abschnitte geben einen kurzen Überblick über Stand und Planung.

4.1 Grundsätzliche Überlegungen

Zwischensprachen-Konzept. Um die Programmierung verteilter und heterogener Plattformen zu ermöglichen ist einerseits eine Unabhängigkeit der Programmiersprache von Betriebssystem und Hardwareplattform erstrebenswert. Andererseits soll Raum gelassen werden für die Entwicklung alternativer, komfortabler Hochsprachen, die jedoch die oben beschriebenen Grundkonzepte unterstützen und miteinander kombinierbar sein sollen. Daher baut die STAGE-Plattform auf einer betriebssystem- und prozessorunabhängigen Zwischensprache (STAGE intermediate language, STAGE IL) auf, in der Anwendungen *installiert* werden (vergleichbar dem Java Byte Code bzw. der Microsoft Intermediate Language des .NET-Frameworks). In diese Sprache werden Programme übersetzt, nachdem sie in einer Hochsprache geschrieben wurden. Als erste "Hochsprache" wird im ersten Ausbau eine einfache, Assembler-nahe Sprache (STAGE ASM) unterstützt, die im wesentlichen die in der Zwischensprache verfügbaren Konstrukte anbietet, siehe auch Kapitel 4.2. Die in Kapitel 2 gezeigten Codeabschnitte wurden in dieser Sprache verfasst.

Interpreter-Ansatz. Für die Ausführung von Programmen wurde auf die Möglichkeit der Übersetzung gänzlich verzichtet, d.h. die virtuelle Maschine (STAGE VM) ist im Wesentlichen ein Zwischensprachen-Interpreter. Dies erschien aus verschiedenen Gründen vorteilhaft, auch wenn die Rechenleistung darunter leidet. Die Einführung eines VM-internen Just-In-Time-Compilers ist erst für spätere Versionen angedacht.

4.2 Plattformüberblick

Abbildung 9 zeigt die Komponenten der STAGE-Plattform im Überblick. Neben der eigentlichen virtuellen Maschine (rechts) sind noch verschiedene Werkzeuge verfügbar oder in Entwicklung:

- *Debugger und VM Manager*
  Ermöglichen Debugging, Starten, Stoppen und Überwachen der virtuellen Maschine.
- *Repository Manager mit Repository*
  Im Repository liegen die für die VM verfügbaren Resources und Bundles. Diese können mittels des Repository Managers zusammengestellt und verwaltet werden. Das Format der im Repository abgelegten Modules bzw. Definitions ist XML-basiert.
- *Editor und Compiler*
  Im ersten Ansatz ist derzeit nur ein Compiler für die bereits erwähnte Assembler-nahe Sprache (STAGE ASM) verfügbar. Dieser wurde teilweise selbst geschrieben, teilweise mit gängigen Werkzeugen generiert. Als Editor genügt zur Zeit ein Texteditor - komfortablere Editoren für weitere Hochsprachen sind vorgesehen.

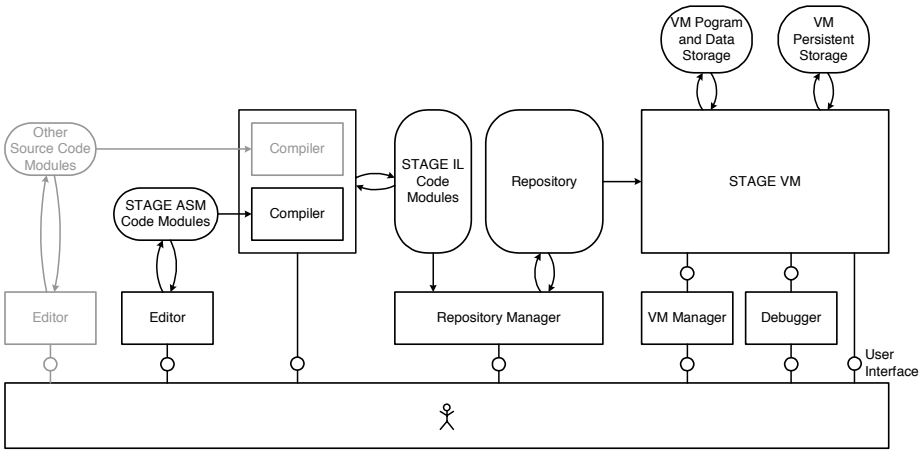


Abbildung 9: STAGE-Plattform

4.3 Virtuelle Maschine

Diese soll prinzipiell auf mehrere Rechner verteilbar sein, d.h. aus einer Menge kooperierender "Knoten" (ein VM Knoten pro Rechner bzw. Prozessor) bestehen. Die folgende Beschreibung bezieht sich auf einen einzelnen Knoten.

Interne Komponenten. Intern besteht jeder VM Knoten im Wesentlichen aus folgenden Komponenten:

- *Code Loader und Runtime Type Repository*
  Benötigte Typdefinitionen werden erst bei Bedarf durch den Code Loader in den

Speicher geladen. Geladene Typdefinitionen werden im Runtime Type Repository verwaltet, sodass bei erneuter Referenzierung eines bereits geladenen Typs dieser dort ausgelesen werden kann.

- *Processing Units*
  Wie bereits oben (siehe Kapitel 2.3) diskutiert, kann ein einzelner Akteur mehrere Operationsfolgen nebenläufig ausführen. Dies betrifft einerseits die Threads, andererseits aber auch nebenläufig angestoßene Operationen auf operationellen Speichern oder Kanälen. Daher ist (konzeptionell) für jeden Ort eine "Processing Unit" vorgesehen, die - nebenläufig zu anderen Processing Units - die zu "ihrem" Ort gehörende Operationsfolge abwickelt.
- *Access Scheduler*
  Die Processing Units sind dann untereinander zu synchronisieren, wenn konkurrierende Zugriffe auf den gleichen Ort durchgeführt werden. Dazu dient der Access Scheduler, bei dem die Processing Units alle Zugriffe auf Orte anmelden. Der Access Scheduler verwaltet und gewährt diese Zugriffe, sodass auf jedem Ort eine Serialisierung der Zugriffe stattfindet. Er stellt dabei die in Kapitel 2.4 vorgestellten Konsistenzbedingungen sicher und koordiniert die implizit definierten Transaktionen bzw. Snapshots (vgl. Kapitel 2.6) - transparent für den Programmierer.

Soft Threading - Processing Unit Multiplex. Eine "naive" Implementierung würde für jede der nebenläufig agierenden Processing Units einen eigenen Betriebssystem-Thread nutzen. Da dies wegen der hohen Kosten für die Bereitsstellung und Synchronisation vieler Threads zu ineffizient wäre, wurde ein eigenes Multithreading ("Soft Threading") realisiert. Dazu existiert in jedem VM Knoten eine einzige, *multiplexfähige Processing Unit*, die von einem *Scheduler* zwischen den verschiedenen Operationsfolgen "umgeschaltet" wird. Abbildung 10 zeigt den vereinfachten inneren Aufbau der VM.

Ein erste Version der virtuellen Maschine wird derzeit fertiggestellt. Im ersten Ausbau ist die VM jedoch noch auf einen Knoten beschränkt.

5 Abschließende Bemerkungen und Ausblick

Der vorgestellte Ansatz kann als "architekturorientiert" bezeichnet werden, da er sich konsequent an den eingangs diskutierten architekturellen Aspekten komplexer Systeme orientiert.

- Erstens bietet die STAGE-Plattform Konstrukte an, die auf die eingangs erwähnten architekturellen Sichten abgestimmt sind und somit auch eine klare Unterscheidung von System- und Softwarekomponenten ermöglichen. Erste Erfahrungen mit Anwendungsbeispielen zeigen, dass sich insbesondere Modelle auf Ebene des Conceptual View sehr direkt beschreiben lassen.
- Als zweites wichtiges Merkmal ist die Berücksichtigung von Verteilungsaspekten, d.h. die Integration der Datenkonsistenzmodelle, zu nennen. Diese basiert

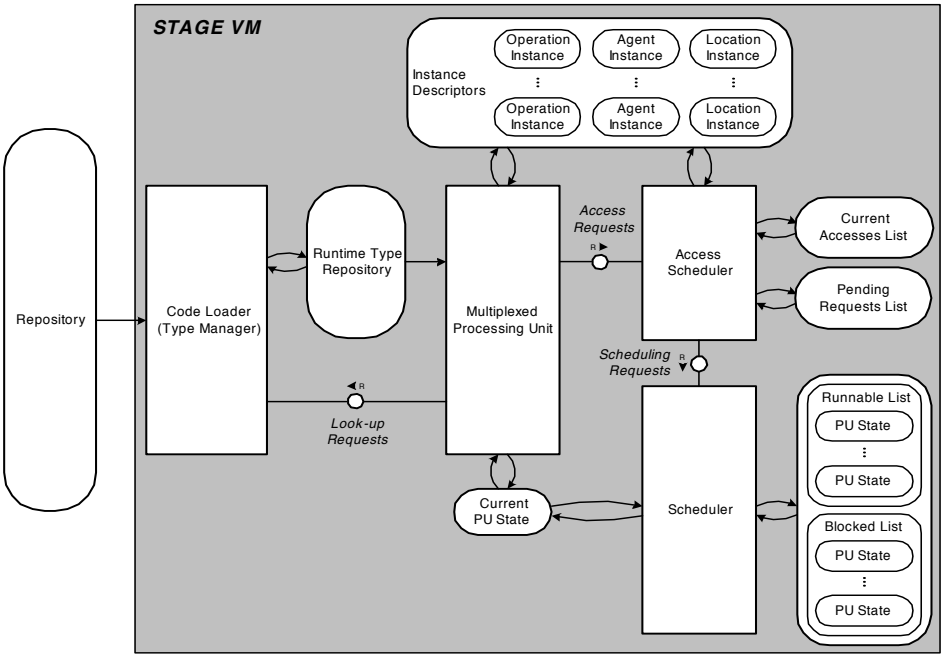


Abbildung 10: STAGE virtuelle Maschine, Einzelknoten

u.a. auf der Unterscheidung von Akteuren und Orten, einem besonderen Merkmal von FMC.

Im Gegensatz zur “Model Driven Architecture” der OMG geht das STAGE-Projekt nicht von existierenden Plattformen aus und ergänzt diese um eine Modellierungsebene, sondern definiert eine Plattform ausgehend von den Modellierungskonzepten. Im Sinne der modellbasierten Entwicklung erscheint es daher nur konsequent, ergänzende Werkzeuge zur Erstellung grafischer Modelle und deren Transformation in die STAGE-Zwischensprache zu entwickeln.

Als laufendes Forschungsprojekt wirft das STAGE-Projekt natürlich noch weitere Fragen auf, die z.B. die Effizienz der virtuellen Maschine betreffen oder auch eine mögliche Umstellung des Repository auf den XMI-Standard (XML Meta-Data Interchange) der OMG.

Literaturverzeichnis

[BCK98] Bass, Clements, Kazman: Software Architecture in Practice. Addison Wesley Longman, 1998

[Bu98] Andreas Bungert: Beschreibung programmierter Systeme mittels Hierarchien intuitiv verständlicher Modelle. Shaker Verlag Aachen 1998

[Cl96] Paul Clements: A Survey of Architecture Description Languages. IEEE Intl. Workshop on Software Specification and Design, 1996

[CL85] Chandy, Lamport: Distributed Snapshots - Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985

[Gr01] Gröne, Knöpfel, Kugel, Schmidt: The Apache Modelling Project. Hasso-Plattner-Institut für Softwaresystemtechnik, Potsdam, www.hpi.uni-potsdam.de/apache, 2001

[GR93] Gray, Reuter: Principles of Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993

[HNS00] Hofmeister, Nord, Soni: Applied Software Architecture, Addison Wesley Longman, 2000

[In89] Inmos Ltd. Occam 2 - Das Referenzhandbuch. Carl Hanser Verlag, 1989

[Ke03] Frank Keller: Über die Rolle von Architekturbeschreibungen im Software-Entwicklungsprozess. Dissertation, Hasso-Plattner-Institut an der Universität Potsdam, 2003

[KT02] Keller, Tabeling et. al. Improving Knowledge Transfer at the Architectural Level - Concepts and Notations. The 2002 International Conference on Software Engineering Research and Practice, Las Vegas USA, 2002

[KW03] Frank Keller, Siegfried Wendt: FMC - An Approach Towards Architecture-Centric System Development. Proceedings of 10th IEEE Symposium and Workshops on Engineering of Computer Based Systems, Huntsville USA, 2003

[Mo97] Robert T. Monroe et. al.: Architectural Styles, Design Patterns, and Objects. IEEE Software, Vol. 14, No. 1, 1997

[MT97] Medvidovic, Taylor: A Classification and Comparision Framework for Software Architecture Description Languages. Dept. of Information and Computer Science, University of California, Irvine, 1997

[Mu93] Sape Mullender (Ed.): Distributed Systems. Addison Wesley, 1993, pp. 55

[Sh95] Mary Shaw et. al.: Abstractions for Software Architecture and Tools to Support Them. Computer Science Dept. Carnegie Mellon University, Pittsburg PA, March 1995

[Sz02] Clemens Szyperski: Component Software. Addison Wesley, 2nd Ed., 2002

[Ob04] Object Management Group: Unified Modeling Language: Superstructure. Version 2.0, final adopted specification, www.omg.org/uml, Stand im Juni 2004

[Ta00a] Peter Tabeling: Der Modellhierarchieansatz zur Beschreibung nebenläufiger, verteilter und transaktionsverarbeitender Systeme. Shaker Verlag, Aachen 2000

[Ta00b] Peter Tabeling: Der Modellhierarchieansatz zur Beschreibung nebenläufiger und verteilter Systeme. Beitrag zum GI-Workshop „Visuelle Verhaltensmodellierung verteilter und nebenläufiger Systeme“ Münster, 2000

[Ta02a] Peter Tabeling: Ein Metamodell zur architekturorientierten Beschreibung komplexer Systeme. Modellierung 2002 - Arbeitstagung der Gesellschaft für Informatik, Lecture Notes in Informatics - Proceedings, 2002

[Ta02b] Peter Tabeling: Multi-level Modeling of Concurrent and Distributed Systems. The 2002 International Conference on Software Engineering Research and Practice, Las Vegas USA, 2002

[Ta04a] Peter Tabeling: Architectural Description with Integrated Data Consistency Models. IEEE International Conference and Workshop on the Engineering of Computer-Based Systems 2004, Brno, IEEE Proceedings, May 2004

[Ta04b] Peter Tabeling: The STAGE Project Homepage. Webseiten, URL: stage.hpi.uni-potsdam.de, Stand vom August 2004

[We04] Siegfried Wendt et. al: The Fundamental Modeling Concepts Homepage. Webseiten, URL: fmc.hpi.uni-potsdam.de, Stand vom Juli 2004

[We82] Siegfried Wendt: Der Kommunikationsansatz in der Software-Technik. data report 17 (1982) Heft 4

[We89] Siegfried Wendt: Nichtphysikalische Grundlagen der Informationstechnik - Interpretierte Formalismen. Springer Verlag, Heidelberg 1989