Skriptanbindung für das HADES Simulations-Framework

vorgelegt von

Matthias Meyer

Erstbetreuer: Dr. Norman Hendrich
Arbeitsbereich Technische Grundlagen der Informatik (TECH)

Zweitbetreuer: Prof. Dr.-Ing. Bernd Page
Arbeitsbereich Angewandte und Sozialorientierte Informatik (ASI)

13. Juni 2000

Diplomarbeit am Fachbereich Informatik, Arbeitsbereich TECH, Universität Hamburg

Erklärung

Ich versichere, daß ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und daß ich diese Arbeit in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Hamburg, den 13. Juni 2000

(Matthias Meyer)

Danksagung

Diese Diplomarbeit ist am Arbeitsbereich Technische Grundlagen der Informatik (TECH) des Fachbereichs Informatik der Universität Hamburg entstanden. Mein besonderer Dank gilt Dr. N. Hendrich für seine Unterstützung meiner Arbeit, speziell seine Kenntnisse des HADES Frameworks waren vor allem bei der Implementierung wichtig. Für die weitere Betreuung durch Prof. Dr.-Ing. Bernd Page möchte ich mich ebenfalls bedanken, sowie bei meinem privaten Umfeld, für dessen Unterstützung und Verständnis.

Zusammenfassung

Diese Arbeit erweitert das HADES-Simulations-Framework des Fachbereiches Informatik der Universität Hamburg um die Anbindung einer Skriptsprache. Verschiedene Skriptsprachen und deren Zugriff auf Java wurden evaluiert und Python, in seiner Java-Variante JPython, für die Realisierung ausgewählt.

Durch die entstandene Skriptanbindung erhält das Simulations-Framework mehr Flexibilität und Funktionalität. Erstens werden bestehende Grenzen der graphischen Benutzeroberfläche mittels einer Java-Python-Shell gelöst. Zweitens ermöglichen die Basismodule der Skriptanbindung das Treiben und Testen von Simulationen. Drittens können neue Komponenten, sowie komplette Designs in Python implementiert werden. Der Entwicklungszyklus verkürzt sich gegenüber der Java-Realisierung, es entfällt der Übersetzungsschritt, und die Komponenten können interaktiv in der Java-Python-Shell getestet werden.

Dieses wird anhand der Entwicklung einer asynchronen Micropipeline demonstriert. Benutzer können das Hamburg-Design-System nach ihren Bedürfnissen anpassen und um neue Simulationskontexte erweitern, ohne das Veränderungen am Systemkern vorgenommen werden müssen. JPython-Modulen steht die komplette Systemfunktionalität zur Verfügung und sie können frei programmiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Grundlagen und Einführung

Kapitel 1

Einleitung

Dieses Kapitel begründet das Vorhaben dieser Diplomarbeit und setzt sie in einen thematischen Gesamtzusammenhang. Abschließend wird die genaue Zielsetzung und die Gliederung der Diplomarbeit erläutert.

1.1 Motivation

Die graphische Oberfläche einer Applikation (GUI) stellt dem Benutzer eine gute Interaktion mit der Anwendung zur Verfügung. Auf die Aktionen des Benutzers wird direkt reagiert und er erhält Zugriff auf die Funktionalität des Systems. Problematisch wird dieses starre Konzept, wenn z.B. der Benutzer bestehende Arbeitsschritte verändern oder gänzlich neue dem System hinzufügen möchte. Solche Änderungswünsche von Benutzern führen oft zu einer Systemanpassung durch den Hersteller der Applikation, dieses kann jedoch nicht für jeden Änderungswunsch geschehen. Ein besseres Konzept ist es dem Benutzer die Möglichkeiten einzuräumen, die Applikation auf seine Bedürfnisse anzupassen. Die Einbettung einer Skriptsprache (s. Seite 10) in die bestehende Applikation erweitert deren Flexibilität, ohne daß Änderungen am Systemkern vorgenommen werden müssen. Der Benutzer kann die Applikation durch das Erstellen eigener kleiner Programme (Skripte oder Module) leicht manipulieren oder erweitern. Sich wiederholende Arbeitsabläufe können somit gespeichert werden und stehen danach als neue Systemfunktionalität zur Verfügung. Von Seiten des Herstellers muß in die Applikation nicht eingegriffen werden. Skripte können gegebenenfalls zu Systempaketen zusammengefaßt und anderen Benutzern zugänglich gemacht werden.

Am Arbeitsbereich für Technische Grundlagen der Informatik (TECH) des Fachbereiches Informatik der Universität Hamburg wird seit 1997 das Hamburg-Design-System (kurz HADES) entwickelt [Hen98]. Hierbei handelt es sich um ein Simulations-Framework für die Logiksimulation und Systemsimulation integrierter Schaltungen. Dessen graphische Benutzeroberfläche stößt bei komplexen hierarischen Simulationsmodellen an seine Grenzen. Das Anwendungsspektrum von HADES soll durch eine Skriptanbindung dynamisch vergrößert werden. Der Anwender soll interaktiv auch mit komplexen Simulationen interagieren können. Für den Einsatz von HADES in der Lehre ist dies ein wichtiger Vorteil gegenüber teuren kommerziellen Simulatoren.

HADES gewinnt durch seine Skriptanbindung an Flexibilität, Funktionalität und Benutzerfreundlichkeit. So ist es z.B. möglich, mittels einfacher Shell-Kommandos Gatter

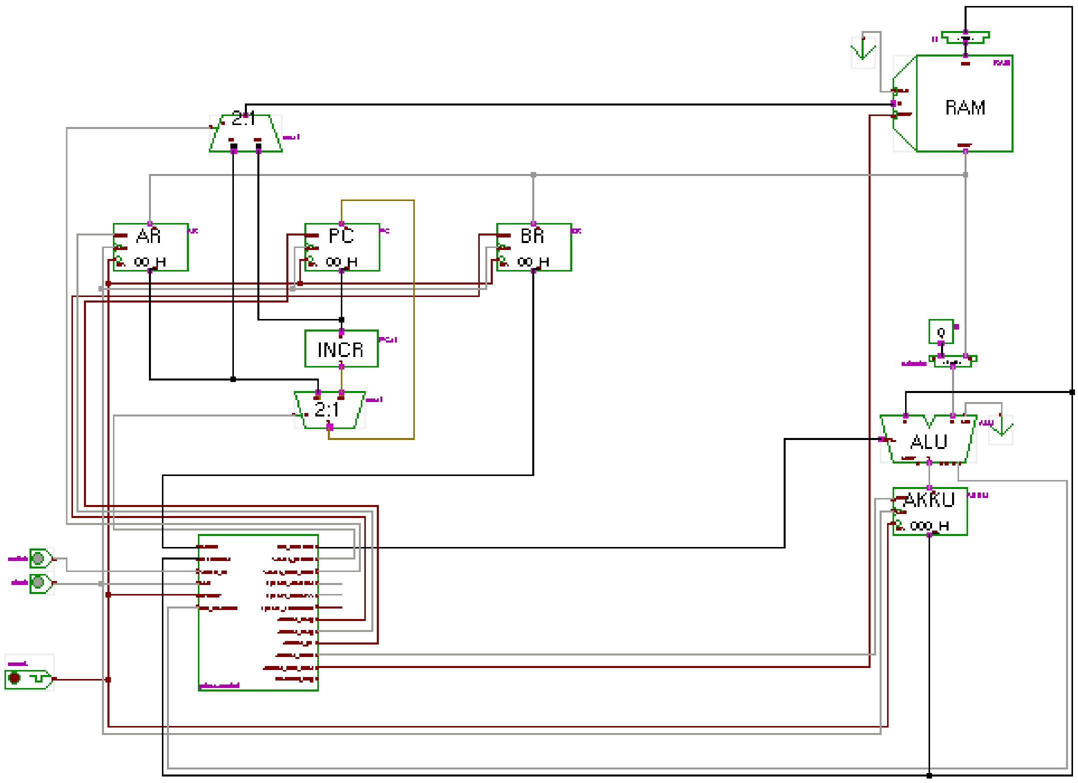

Abbildung 1.1: Das Design der 'PRIimitiven MAschine' PRIMA stellt einen minimalen Von-Neumann-Rechner dar, auf der rechten Seite befindet sich das Rechenwerk und auf der linken das Steuerwerk. Diese Darstellung des HADES prima-Designs ist durch die Steuerleitungen unübersichtlich, die Datenleitungen sind schwer zu erkennen.

kompletter Subdesigns direkt zu manipulieren. Dies ist im Prinzip über das GUI ebenfalls möglich, jedoch muß hierbei für jedes Subdesign ein neuer HADES-Editor geöffnet und jedes Gatter einzeln manipuliert werden. Bei komplexeren Simulationen is dies praktisch unmöglich. Eine komfortable und mächtige Shell-Programmierung dagegen erleichtert und beschleunigt solche Arbeitsabläufe.

Andere Manipulationen an einem HADES-Design werden überhaupt erst durch die Skriptanbindung möglich, z.B um die Signalläufe des `prima`-Designs (s. Abb. 1.1) besser zu visualisieren, können alle Steuerleitungen bis auf ihren Anfang und ihr Ende reduziert werden (s. Abb. 1.2). Hierfür wurden entsprechende Prozeduren einmal in einem Modul zusammengefaßt und in der kommandozeilenorientierten Shell der Skriptanbindung ausgeführt. Das HADES-System wurde somit vom Benutzer um neue Funktionalität erweitert, ohne das dieser das System manipulieren oder übersetzen mußte. Solche Module können interaktiv zur Laufzeit erstellt und getestet werden.

Neue Simulationskomponenten können in Form von kurzen Skripten zur Laufzeit erstellt und direkt getestet werde. Der Entwickluszyklus dieses `try-and-error`-Verfahrens

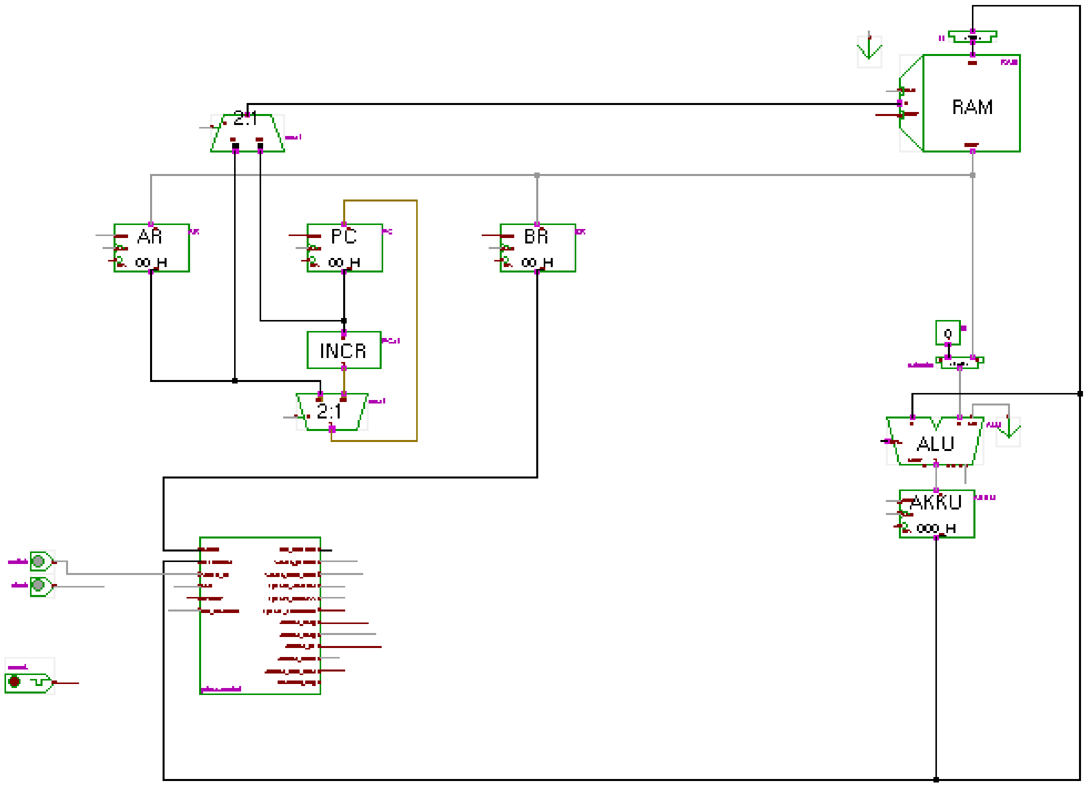

Abbildung 1.2: Nach dem Ausblenden der Steuerleitungen sind die Datenleitungen und somit das gesamte prima-Design wesentlich deutlicher dargestellt. Dieses wurde durch eine einfache Funktion eines importierten Benutzer-Moduls der Skriptanbindung realisiert. Der HADES-Editor bietet diese Funktionalität nicht.

ist deutlich kürzer als die Komponentenimplementierung in Java [GJS96] [Fla98] [HC99] [Hen97b]. Es entfällt der Übersetzungsschritt, die Aktualisierung der Java-Klassen und ein Neustart des Systems. Die Skripte der Komponentenerweiterung sind bei identischer Funktionalität und Arbeitsweise kürzer als ihre Java-Versionen. Für Simulationsexperimente ist es wichtig, daß deren Stimuli vorher spezifiziert werden können. Zur angegebenen Simulationszeit werden diese Stimuli an die entsprechende Komponente übergeben und die Erwartungswerte bestimmter Signale können geprüft werden. Simulationsexperimente können einfach über die graphische Benutzeroberfläche angelegt werden. Deren Erstellung mittels frei programmierbaren Modulen bietet dem Benutzer jedoch mehr Möglichkeiten, z.B. können Pipelines beliebiger Länge, als Funktion realisiert werden. Insbesondere stehen die neuen Simulationskomponenten der Skriptanbindung den Benutzer-Modulen direkt zur Verfügung, das HADES-GUI unterstützt diese nicht und muß gegebenenfalls angepaßt werden.

1.2 Zielsetzung der Arbeit

Das Hauptziel dieser Arbeit ist es, das HADES-Framework um eine Skriptsprache zu erweitern. Diese leistungsstarke, vollwertige Skriptsprache muß mit dem bestehenden Java-System harmonieren und es sinnvoll ergänzen.

Im theoretischen Teil sind die bestehenden Skriptsprachen auf deren mögliche Java-Anbindung hin zu untersuchen. Die gegebenenfalls vorhandenen, in Java realisierten, Skriptsprachen-Interpreter sind zu evaluieren. Die Auswahl einer geeigneten Skriptsprache für die Realisierung der technischen Teilaufgaben ist daher das erste Teilziel dieser Arbeit. Sollte keine geeignete Skriptsprachenlösung existieren, ist über eine eigene Sprachimplementierung nachzudenken.

Ein allgemein gültiges Konzept einer Skriptanbindung ist zu entwickeln und anhand einer Referenzimplementierung für HADES zu überprüfen. Der HADES-Systemkern darf dabei nicht durch diese Arbeit verändert werden, er ist als fremdverwaltete Applikation anzusehen. Eine Skriptanbindung muß trotz des interaktiven Charakters der Anwendung realisiert werden. Im praktischen Teil ist zusätzlich die folgende Funktionalität als Mehrwert für das HADES-System zu Implementieren. Hierbei sollen die Möglichkeiten der gewählten Skriptsprache und des Interpreters genutzt und gegebenenfalls Schwachstellen aufgezeigt werden.

- Realisierung einer interaktiven Benutzer-Shell
  Das HADES-System soll in seiner Gesamtheit durch eine interaktive Benutzer-Shell steuer- und manipulierbar sein. Hierfür muß ein umfassender Zugriff auf die entsprechenden Java-Objekte von HADES realisiert werden. Diese Shell ist mit zugehöriger graphischer Oberfläche zu implementieren und in die bestehende Systemumgebung zu integrieren. Sie muß im interaktiven Einsatz umfassende Funktionalität für die Bearbeitung von HADES-Designs sowie die Möglichkeiten gängiger UNIX-Shells bieten. Diese Shell bildet die Ausführungsumgebung der folgenden beiden Teilziele.

- Erstellung einer HADES-STL
  Aufbauend auf dem ersten Teilziel sind die Möglichkeiten der Simulations- und Test-Sprache (STL) des Cadence-Design-Systems [Cad97] für HADES zu adaptieren. Geeignete Basis-Module sollen die Erstellung einfacher STL-Skripte gewährleisten. Diese sollen das Anlegen von Stimuli und die Prüfung von Erwartungswerten eines HADES-Simulationsexperimenentes ermöglichen.

- Entwicklung neuer Simulationskomponenten
  Während die ersten beiden technischen Teilziele manipulierend und steuernd in das HADES-System eingreifen, sind in diesem Teil neue Simulationskomponenten durch die Skriptanbindung zu realisieren. Deren Entwicklung soll entsprechend der Erzeugung neuer HADES-Komponenten in Java geschehen [Hen97]. Das entwickelte Konzept soll am Beispiel von asynchronen Komponenten überprüft werden. Diese Simulationsobjekte müssen sich in das HADES-System integrieren lassen und dürfen dieses in seiner Funktionalität nicht einschränken.

Benutzung spezieller Begriffe

Viele der in dieser Arbeit benutzten Begriffe besitzen je nach Kontext unterschiedliche Bedeutungen. Um einem Mißverstehen entgegenzuwirken, wird durchgehend versucht, den

Gebrauch von Anglizismen zu vermeiden. Mehrheitlich wird auf den deutschen Begriff zurückgegriffen. Bei etablierten englischen Begriffen ohne deutsches Pendant wird der englische Ausdruck verwendet.

1.3 Gliederung der Arbeit

Die Arbeit gliedert sich in 11 Kapitel, die auf vier Teile aufgeteilt sind. In einem fünften Teil sind die Anhänge zu finden.

Teil I - Grundlagen und Einführung
Dieser Teil begründet das Gesamtvorhaben und führt den Leser nach einer allgemeinen Darstellung des Standes der Wissenschaft in das spezielle Umfeld dieser Arbeit ein.

Kapitel 1 - Einleitung: Dieses Kapitel erklärt das Vorhaben dieser Diplomarbeit und setzt sie in einen thematischen Gesamtzusammenhang. Abschließend wird die genaue Zielsetzung und die Gliederung der Diplomarbeit erläutert.

Kapitel 2 - Skriptsprachen: Dieses Kapitel wird durch die Diskussion der beiden Begriffe Systemprogrammiersprache und Skriptsprache bestimmt. Nachdem die Vor- und Nachteile aufgezeigt wurden, findet eine Erörterung der Vorteile eines gemischten Systems statt.

Kapitel 3 - Simulation und Simulationssteuerung: Dieses Kapitel führt in die Grundlagen der Simulation und der Simulationssteuerung von HADES ein. Für diese Arbeit wichtige Teilaspekte des HADES-Frameworks werden explizit dargestellt.

Teil II - Evaluation geeigneter Skriptsprachen und deren Interpreter
Dieser Abschnitt beschreibt den Aufbau der Untersuchung. Geeigneten Skriptsprachen werden untersucht und eine ausgewählt. Die in Java implementierten Interpreter dieser Skriptsprachen werden evaluiert und anhand der bestehenden Anforderungen bewertet.

Kapitel 4 - Aufbau der Untersuchung: Dieses Kapitel erarbeitet die Anforderungen, nach denen die Skriptsprachen bezüglich ihrer Java-Anbindungen untersucht werden sollen. Die geeigneten Skriptsprachen werden kurz vorgestellt.

Kapitel 5 - Evaluation der Interpreter: Dieses Kapitel evaluiert die in Java realisierten Interpreter der geeigneten Skriptsprachen. Nach einer Einführung in die entsprechende Java-Anbindung wird auf die Implementierung der einzelnen Interpreter eingegangen und die Abschnitte durch eine Bewertung abgeschlossen.

Kapitel 6 - Bewertung der Java-Anbindungen: Dieses Kapitel faßt die Untersuchung in Form einer abschließenden Bewertung zusammen und begründet die Auswahl von JPython. Sie wird durch eine vergleichende Betrachtung von Python und den anderen Sprachen unterstützt.

Teil III - Realisierung der HADES-Skriptanbindung
Dieser Teil beschreibt den Entwurf und die Realisierung der Skriptanbindung des HADES-Frameworks. Sowohl das Gesamtkonzept, als auch das der interaktiven Skriptsteuerung und das der Simulations- und Test-Sprache von HADES und deren Realisierung werden dargestellt. Ebenfalls wird in einem separaten Kapitel auf die Komponentenerweiterung von HADES durch Benutzer-Module eingegangen.

Kapitel 7 - Konzeption der Skriptanbindung: Dieses Kapitel stellt konzeptionell die Grundidee einer Skriptanbindung dar. Anschließend wird die Gesamtarchitektur der HADES-Skriptanbindung vorgestellt es werden die Konzepte der interaktiven Skriptsteuerung und der Komponentenerweiterung besprochen.

Kapitel 8 - Realisierung der interaktiven Skriptsteuerung: Dieses Kapitel dokumentiert die Umsetzung der interaktiven Skriptsteuerung durch die Java-Python-Shell (jpshell) und dessen Funktionsumfang. Auf die Vorstellung ihrer Funktionalität und Implementierung folgt eine Darstellung der Möglichkeiten der Java-Python-Shell anhand ausgewählter Verwendungsbeispiele.

Kapitel 9 - Realisierung der HADES Simulations- und Test-Sprache: Dieses Kapitel dokumentiert die Umsetzung der Simulations- und Test-Sprache für das HADES-Framework. Die Implementierung des Konzeptes sowie dessen funktionales Potential werden dargestellt. Ein ausführliches Beispiel verdeutlicht die Verwendung der HADES-STL.

Kapitel 10 - Realisierung der HADES-Komponentenerweiterung: In diesem Kapitel wird die Erweiterung von HADES um neue Simulationsobjekte dargestellt. Diese werden anstatt in Java, durch JPyton-Module implementiert. Ebenfalls wird die Erzeugung von HADES-Design mittels einfacher JPyton-Module vorgestellt. Dieses geschieht anhand des Themenkomplexes der asynchronen Komponenten.

Teil IV - Resümee
In diesem Teil wird eine zusammenfassende Rückschau auf die erarbeiteten Ergebnisse geleistet. Die Ergebnisse werden vor dem Hintergrund der Aufgabenstellung diskutiert und bewertet. Abschließend werden Perspektiven und Ansätze für weiterführende Arbeiten dargestellt.

Kapitel 11 - Bewertung und Ausblick: Dieses Kapitel blickt zusammenfassend auf die Ergebnisse der Diplomarbeit zurück und bewertet diese im Rahmen der Aufgabenstellung. Abschließend werden Perspektiven und Ansätze aufgezeigt, die in dieser Arbeit nicht mehr bearbeitet werden konnten.

Teil V - Anhänge Dieser Teil beinhaltet die folgenden Anhänge.

Plattformtest von JPython: Dieser Anhang dokumentiert das Verhalten von JPython unter verschiedenen Betriebssystemen und Java-Versionen.

Quellen: Dieser Anhang umfaßt das vollständige Literaturverzeichnis dieser Diplomarbeit. Die Literaturverweise sind in der Arbeit als Marginaltexte neben den zugehörigen Textstellen angeführt.

Kapitel 2

Skriptsprachen

2.1 Einleitung

Jedesmal wenn eine neue Programmiersprache das Interesse auf sich zieht, wie z.B. im Falle von Java, so keimt teilweise in der Informatik die Hoffnung auf, daß es die eine Sprache ist, in der zukünftig alle Programme geschrieben werden. Betrachtet man aber die Vergangenheit, so ist dies niemals geschehen. Weder PL/1 [Sch84] noch C++ [Str92] erfüllten diese Erwartung. Aber die Sehnsucht bleibt bestehen, daß es vielleicht die nächste Sprache sei, zu hoch sind die Kosten für das Erlernen verschiedenster Programmiersprachen für unterschiedlichste Aufgaben [Ros98].

Vielleicht ist diese Hoffnung unbegründet und es wird nie eine Sprache geben, die alle Bereiche abdeckt. Davon ausgehend könnte nun eine abgeschwächte Form darin bestehen, daß es für jeden Aufgabenbereich eine Sprache gibt und der Programmierspezialist nur die eine, für seinen Bereich gedachte, Sprache erlernen muß. Dies würde jedoch zu einer großen Anzahl von Spezialsprachen führen. Eine Alternative wäre die Verwendung von mehreren Sprachen, die zusammen mehr abdecken, indem sie sich ergänzen. Dieser Gedankengang ist nicht neu, denn die Designer von UNIX verfolgten genau diese Vision. Sie realisierten ihr System von Anfang an durch zwei Sprachen, mit C [KR88] als Systemprogrammiersprache (s. Seite 10) für die Entwicklung von Applikationen und Tools, sowie der Shell für das Skripting und zum Verbinden von C-Programmen. Die Shell wurde später von Skriptsprachen wie Tcl [Ous94], Perl [WCSP96] und Python [Ros2000] ein wenig verdrängt, aber das Prinzip der zwei sich ergänzenden Sprachen blieb erhalten. Auch Microsoft setzt auf die Mixtur aus zwei Sprachen. Sie realisierten ihren Systemkern mit C++ und ergänzten diesen um VisualBasic [MS95], das sehr einfach auf C++-Controls zugreifen kann.

Beide Sprachen ergänzen sich, und für spezielle Aufgaben empfiehlt sich meist eine von ihnen. Das Hinzunehmen einer Skriptsprache hat den großen Vorteil, daß die gesamte Applikation an Flexibilität gewinnt und späteres Konfigurieren wesentlich erleichtert wird. “Interpreters also make applications more flexible by allowing users to program the applications at run-time. For example, many synthesis and analysis tools for integrated circuits include a Tcl interpreter; users of the programs write Tcl scripts to specify their designs and control the operation of the tools.” [Ous98].

Skriptsprachen sind leicht zu lernen, portabler als Systemsprachen und ihre Programme sind bemerkenswert kurz. Ihr Hauptvorteil liegt aber in ihrer leichten Erweiterbarkeit des *scripting frameworks*. Der Begriff *scripting framework* umfaßt das gesamte Skriptpaket, bestehend aus der eigentlichen Skritsprache, dem Interpreter und den vorgesehenen Erweiterungsmechanismen. Die Basis einer Skriptsprache ist ihr Interpreter, der meist in einer Systemprogrammiersprache implementiert ist. Bei ihrer Entwicklung steht die leichte Erweiterbarkeit im Vordergrund. Einige in dieser Arbeit untersuchten freien Skriptsprachen (Scheme [DD96] [CR98], Tcl und Phyton) wurden und werden durch ihre "Open-Source"-Gemeinde stetig erweitert, so daß sie heute mächtige und sehr ernst zu nehmende Programmiersprachen darstellen. Jede von ihnen hat eine blühende Fangemeinde, die sich um deren Erweiterung kümmert und meist eine Person, mit der die Sprache assoziiert wird und die über den eigentlichen Sprachkern wacht.

2.2 Diskussion der Begriffe Systemprogrammiersprache und Skriptsprache

Eine genaue und feste Differenzierung zwischen Skript- und Systemprogrammiersprachen kann nicht existieren. Es ist ein Fakt, daß einige der besten Programmierumgebungen es geschafft haben, Hochsprachen wie zum Beispiel Lisp oder Smalltalk in allen Ebenen einer Systemarchitektur einzusetzen. Sie sind auf der Komponentenebene eindeutig eine Systemprogrammiersprache, andererseits verketten sie bestehende Programme zu völlig neuen Applikationen oder werden als interaktive Debugger eingesetzt, was sie eindeutig zu den Skriptsprachen zählen läßt. Auch wenn die Programmiersprachen nicht eindeutig eingeteilt werden können, so können Anwendungssituation und Problemstellungen doch recht deutlich einer Sichtweise zugeordnet werden. Bietet sich für ein Problem eine Skriptlösung an, so sollte eine Sprache gewählt werden, die alle Eigenschaften hierfür mitbringt. Genauso sollten für große Architekturen und deren Komponenten die Vorteile einer klassischen Systemprogrammiersprache, wie zum Beispiel ihr deutlich statischeres Typensystem genutzt werden. Die Grundprämisse muß sein, die richtige Sprache für das jeweilige Problem zu nehmen.

2.2.1 Begriffsbestimmung: Skriptsprache

Der Begriff Skriptsprache steht im allgemeinen für eine interpretierte, spät bindende und schwach typisierte Programmiersprache, die entwickelt wurde, um bestehende Programme zusammenzubringen (*glue language*). Ihre bekanntesten Vertreter sind Tcl, Perl, Python und VisualBasic.

2.2.2 Begriffsbestimmung: Systemprogrammiersprache

Systemprogrammiersprachen werden in der Komponentenentwicklung eingesetzt. Durch ihre Typensicherheit, ihre komplexen Datenstrukturen und durch mächtige Übersetzer und Debugger können mit ihnen riesige Softwareprojekte gemeistert werden. Die klassischen Vertreter sind Sprachen wie C, C++ und Java.

2.2.3 Java als Systemprogrammiersprache von HADES

In dieser Arbeit wird die Sprache Java nicht erneut vorgestellt, sondern auf die Literatur verwiesen [GJS96] [Fla98] [Hen97b] [HC99]. In diesem Abschnitt wird auf Java gesondert eingegangen, weil das gesamte HADES-Framework in Java implementiert wurde.

Java wird in dieser Arbeit als Systemprogrammiersprache angesehen, seine strenge Typisierung ist hierfür ein deutlicher Indikator. Eine Verbesserung gegenüber C++ ist die direkte Unterstützung des "high-level" Datentyps String von Java. Ein weiterer Vorteil ist die schnellere Entwicklung durch Java, denn die Java-Programme sind in der Regel kürzer als ihre C++-Varianten. Ihre Übersetzung ist aufgrund der einfacheren Java-Struktur wesentlich schneller.

Entscheidende Vorteile von Java gegenüber C++ sind die Portabilität des Java-Bytecodes und die automatische Speicherverwaltung von Java (*garbage collection*). Java-Programme werden in einen maschinenunabhängigen Bytecode übersetzt, der dann von der *Java-Virtual-Machine* (JVM) [LY97] [MD97] ausgeführt wird. Diese Bytecode-Interpretation läßt streng genommen Java zu einer Mischsprache werden, deren Einordnung vom Anwendungskontext abhängt.

Durch die Auswahl von Java als Systemprogrammiersprache, stellt sich in dieser Arbeit die Frage, welche Skriptsprache am besten geeignet ist. Durch die *Java-Reflection-API* [Sun97b] hat sich Java gegenüber anderen Sprachen deutlich geöffnet und eine mächtige Anbindung von Skriptsprachen wurde durch sie erst ermöglicht. In Java implementierte Skriptspracheninterpreter können dynamisch Objekte in der virtuellen Maschine von Java erzeugen und vernichten. Zur Laufzeit können die Felder und Methoden von Objekten ermittelt und auf sie zugegriffen werden.

2.2.4 Vorteile von Skriptsprachen

In der Programmentwicklung bestechen Skriptsprachen durch den deutlich kürzeren Quelltext gegenüber den Systemprogrammiersprachen, ohne dabei an Lesbarkeit zu verlieren. Ein wichtiger Grund hierfür ist ihre dynamische und schwache Typisierung, die es ermöglicht, Anweisungen und beliebige Daten in einer Zeile zu kombinieren. Der Mangel an statischer Typsicherheit impliziert auch Risiken, denen jedoch die Vorzüge einer dynamischen Sprache gegenüberstehen und die durch ein geeignetes Fehlerbehandlungskonzept (siehe z.B. die Ausnahmebehandlung in Python) reduziert werden können.

Die meisten objektorientierten Systemprogrammiersprachen, die angeblich ein statisches Typenkonzept haben (z.B. Java), neigen dazu, es mittels unsicherer Typenumwandlung zu durchbrechen, ohne jedoch die schönen Seiten eines dynamischen Konzepts zu bieten. Der bewußte Verzicht auf eine strenge statische Typisierung, wird mit einer gewissen Verlangsamung bezahlt. Dieser Performanceverlust kann durch den größten Vorteil von Skriptsprachen wieder beseitigt werden, ihre einfache Erweiterbarkeit. Ist der Performanceverlust nicht hinnehmbar, so wird der entsprechende Interpreter durch eine Reimplementierung in einer schnelleren Systemprogrammiersprache um diese Funktionalität erweitert. Wird z.B. bei einer Tcl-Applikation festgestellt, daß gewisse Skript-Funktionen zu langsam sind, wird ihre Funktionalität in C reimplementiert und der Tcl-Interpreter um diese neue Funktion erweitert. So wird im Weiteren immer direkt der C-Maschinencode aus den Skripten heraus aufgerufen.

Durch den Einsatz von interpretierten Skriptsprachen beschleunigt sich der Entwicklungszyklus enorm. Die entscheidenden Faktoren sind hauptsächlich das komplette Entfallen der Übersetzungs- und Verbindungsschritte und die Möglichkeit der interaktiven Entwicklung in einer Shell der betreffenden Skriptsprache. Diese Eigenschaft der geringen Evaluationszeit führt dazu, daß mit Skriptsprachen mehr Programme, mit geringerer Quelltextlänge, in kürzerer Zeit zu erstellen sind.

Skriptsprachen werden traditionell hauptsächlich in den folgenden Situationen eingesetzt:

- für das *rapid prototyping* von Applikationen
- als Ersatz für Shellskripte
- um die Applikation für den Benutzer einfacher konfigurierbar zu machen
- für die Erstellung von Benutzungsoberflächen, z.B. durch die ToolKit (TK) Erweiterung von Tcl.
- seltener für die Erstellung kompletter (größerer) Projekte

Wegen diesen Fakten werden Skriptsprachen gerne zum schnellen Erstellen von Prototypen eingesetzt. Dieses haftet ihnen leider als Manko an, wodurch sie in der Praxis meist nicht als richtige Sprachen anerkannt werden und ihre Ergebnisse als zu "einfach" empfunden werden. Sie werden gerne für den sogenannten *quick-and-dirty throw-away code* benutzt, aber große Projekte werden ihnen nicht zugetraut. Diese Vorurteile werden in letzter Zeit durch bemerkenswerte Produkte widerlegt, z.B. den Tcl-Webserver [Scr2000].

Kapitel 3

Simulation und Simulationssteuerung

Dieses Kapitel stellt die technischen Grundlagen des HADES-Frameworks vor. Zunächst wird das allgemeine System und dessen Struktur beschrieben. Hierauf folgt eine Erklärung der für den praktischen Teil wichtigen Teilaspekte von HADES.

3.1 Hamburger-Design-System

3.1.1 Kurzvorstellung des HADES-Simulations-Frameworks

Das HADES-System stellt eine benutzerfreundliche Simulationsumgebung für komplexe Logiksimulation von integrierten Schaltungen dar. Das HADES-System wird in der Lehre der Universität Hamburg eingesetzt und Studenten können ohne eine lange Einarbeitungsphase mit ihm arbeiten. Durch die Portabilität des gesamten Systems, kann dieses auch über das Internet benutzt werden. Die Besonderheit von HADES ist die gelungene Verbindung von Komplexität der Simulationsexperimente, einer einfachen interaktiven Bedienung und den Spracheigenschaften von Java.

In der folgenden Aufzählung werden die wichtigsten Eigenschaften von HADES vorgestellt:

- HADES bietet ein flexibles und erweiterbares Simulations-Framework.
- HADES wurde komplett in Java realisiert und ist im Quelltext frei verfügbar.
- Innerhalb gängiger Web-Browser kann HADES als Applet genutzt werden.
- Simulationen können mittels einer graphischen Oberfläche interaktiv manipuliert werden.
- Unterstützt werden alle Abstraktionsebenen, von der Systemebene hinunter bis zur Gatterebene.
- VHDL-basierte Semantik und das Logikmodel nach IEEE 1164 werden angeboten [IEEE-87].

- HADES-Simulationen können in gängige Simulations-Formate exportiert werden. Ein Austausch von Simulationsexperimenten mit anderen Simulatoren ist möglich.
- HADES-Designs können in ihrer graphischen Darstellung direkt im PostScript-Format gespeichert werden.

Das Anwendungsspektrum eines Simulations-Frameworks ist maßgeblich von den unterstützten Komponenten abhängig. Die HADES-Komponentenbibliotheken umfassen u.a alle Basis-Gatter mit bis zu vier Eingängen, Komplex-Gatter, alle Standard-Flipflops und PIC/MIPS-Conroller sowie Schnittstellen für deren Pheripherie (vollständige Funktionsbeschreibung s. [Hen97]). Das HADES-System verfügt über zusätzliche Editoren, z.B für KV-Diagramme [Meyer98].

3.1.2 Systemarchitektur

Die grundlegende Systemarchitektur von HADES ist in der Abbildung 3.1 dargestellt. Funktional läßt sich HADES in drei Teile aufteilen: den Simulationskern, die graphischen Benutzeroberfläche (HADES-Editor) und das Simulationsexperiment (HADES-Design).

Ein Simulationsexperiment (HADES-Design) besteht aus einer beliebigen Anzahl von Simulationskomponenten (HADES-SimObjects), deren logische Verbindung über separate Objekte (Signale) erfolgt. Die Initialisierung und Interaktion der Komponenten erfolgt über einen ereignisbasierten Simulationskern. Dieser verwaltet in einer sortierten Liste die Simulationsereignisse. Jede Simulationskomponente kann neue Ereignisse erzeugen und gibt sie an den HADES-SimKernel weiter, der diese abhängig von ihrer Simulationszeit in seine Liste einsortiert. Das gesamte HADES-Framework ist in Java implementiert. Dies erlaubt es, die Interaktion zwischen Simulator und den Simulationskomponenten zunächst nur über zwei sehr einfache Java-Interfaces zu definieren, die jederzeit durch Definition weiterer Unterklassen verfeinert werden können. Für den Anwender unmittelbar sichtbar ist die Benutzeroberfläche mit dem Graphikeditor und der Simulationssteuerung. Als Beispiel zeigt die Abbildung 3.3 den Graphikeditor mit dem RT-Modell eines einfachen Mikroprozessors. Bei Bedarf können weitere Fenster geöffnet werden, etwa für die Impulsdiagramme oder die komponenten-spezifischen Editoren.

Der HADES-Editor verfügt über eine Referenz auf das Wurzel-Design der Simulation. Auf die Organisation von HADES-Designs wird im Abschnitt 3.2.1 auf Seite 16 eingegangen. Der Graphikeditor stellt das oberste Design mit seinen zu simulierenden Komponenten dar. Diese graphische Ebene ist von der funktionalen getrennt zu betrachten, diese konzeptionelle Trennung wird im Abschnitt 3.2.2 aufgegriffen. Die Verbindungen zwischen den einzelnen Simulationsobjekten werden über Signale hergestellt. Einen Ausschnitt aus der Klassenhierarchie der HADES Simulationsmodelle zeigt Abbildung 3.2 auf Seite 16. Die `SimObject`-Klasse dient als Basisklasse für alle mit HADES simulierbaren Komponenten. Die `Design`-Klasse enthält Referenzen auf beliebig viele weitere `SimObject`-Komponenten und dient als Abstraktion für ein konkretes Simulationsexperiment. Da ein `Design`-Objekt wiederum andere `Design`-Komponenten enthalten kann, sind hierarchische Modelle beliebiger Tiefe möglich.

Konkrete Simulationsmodelle, etwa für ein XOR-Gatter, werden als Unterklassen von `SimObject` realisiert. Dabei sind weitere Zwischenstufen in der Klassenhierarchie möglich. So repräsentiert die `GenericGate`-Klasse die gemeinsamen Eigenschaften der einzelnen Logikgatter. Über die elementaren Gatter und Komplexgatter hinaus umfassen die Modell-

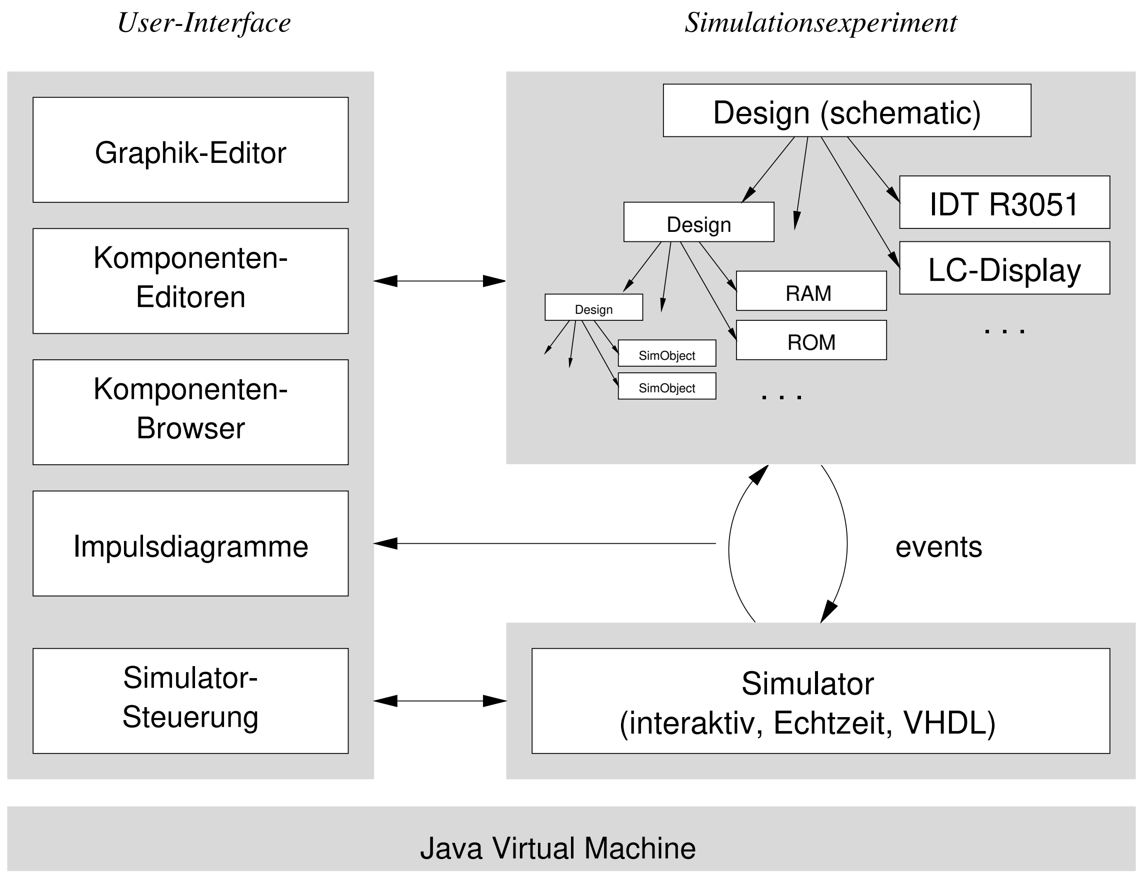

Abbildung 3.1: Überblick über die Systemarchitektur von HADES. Das System gliedert sich in den ereignisbasierten Simulationskern, die Benutzeroberfläche mit ihren Editoren und das Simulationsexperiment.

bibliotheken derzeit auch eine Vielfalt von Modellen für Komponenten auf der Register-Transfer-Ebene, etwa für Addierer, Multiplexer oder Speicher und der Systemebene.

Ein weiterer wichtiger Vorteil von HADES ist die Unterstützung einer interaktiven Simulation, ohne jedoch auf Simulationskomplexität verzichten zu müssen. Simulationsmodelle und Signale können während der Simulation hinzugefügt, gelöscht, oder editiert werden. Dieses macht HADES für die Lehre so wertvoll, Studenten können zu jeder Zeit mit ihren Design interaktiv "herumspielen" und deren dynamisches Verhalten verstehen. Besonders gut wird dieses durch den HADES-Waveformviewer visualisiert, hierbei werden alle Simulationsereignisse aufgezeichnet und graphisch dargestellt. Ausgewählte Signale können explizit durch das Anlegen von HADES-Probes (Abtastpunkte) überwacht werden.

Ein Beispiel für eine Simulation von Einzelkomponenten ist die Simulation des D.CORE [Hen2000a]. Die Einzelkomponenten sind HADES-Modelle der Register-Transfer-Ebene und werden zusammen mit externen Komponenten (Speicher) simuliert (s. Abb. 3.3).

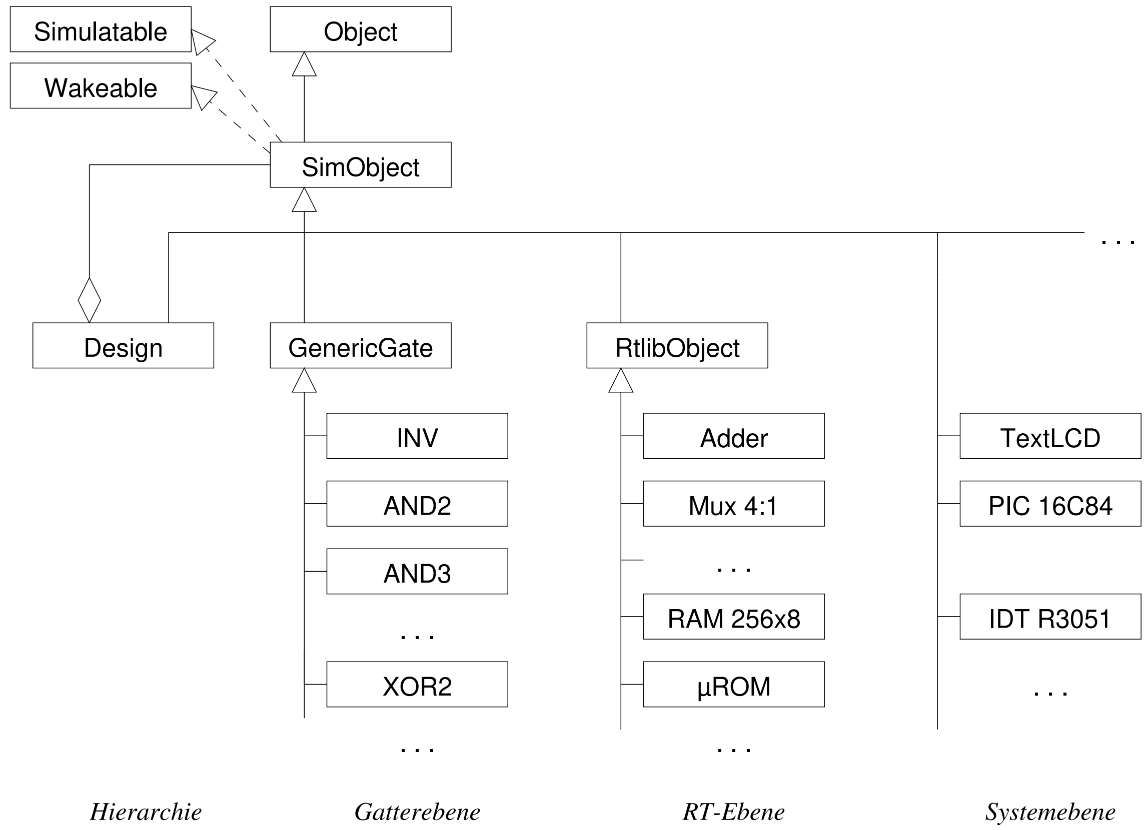

Abbildung 3.2: Dieser Ausschnitt aus der HADES-SimObject Klassenhierarchie verdeutlicht die zentrale Rolle der SimObject-Klasse. Für die einzelnen Ebenen existiert jeweils eine Subklasse. Ein HADES-Design besteht aus einer Menge von SimObject-Objekten und erbt ebenfalls von der SimObject-Klasse. Hierdurch können hierarische Designs gebildet werden.

3.2 Spezielle Teilaspekte

In diesem Abschnitt wird auf Teilaspekte des HADES-Systems eingegangen, sofern diese für das Verständnis des praktischen Teils wichtig sind. Besonders für die Erstellung neuer Komponenten ist ein tieferes Verständnis der Zusammenhänge innerhalb von HADES nötig.

3.2.1 Aufbau und Organisation von HADES-Designs

Die Basisstruktur jeder HADES-Simulation bildet ein Exemplar der `Design`-Klasse. Dieses enthält alle Simulationsobjekte einer Simulation und stellt die Verbindung zum Simulationskern her. Design-Objekte sind ebenfalls Simulationsobjekte und können beliebig verschachtelt werden. Diese hierarischen Designs werden in einer Baumstruktur organisiert, vergleichbar mit dem eines Dateisystems. Ausgehend von einem Wurzel-Design kann in diesem Design-Baum hinabgestiegen werden. Hierbei muß für jedes Subdesign ein neuer HADES-Editor geöffnet werden, der dann dieses Design darstellt. Ab einer gewissen Kom-

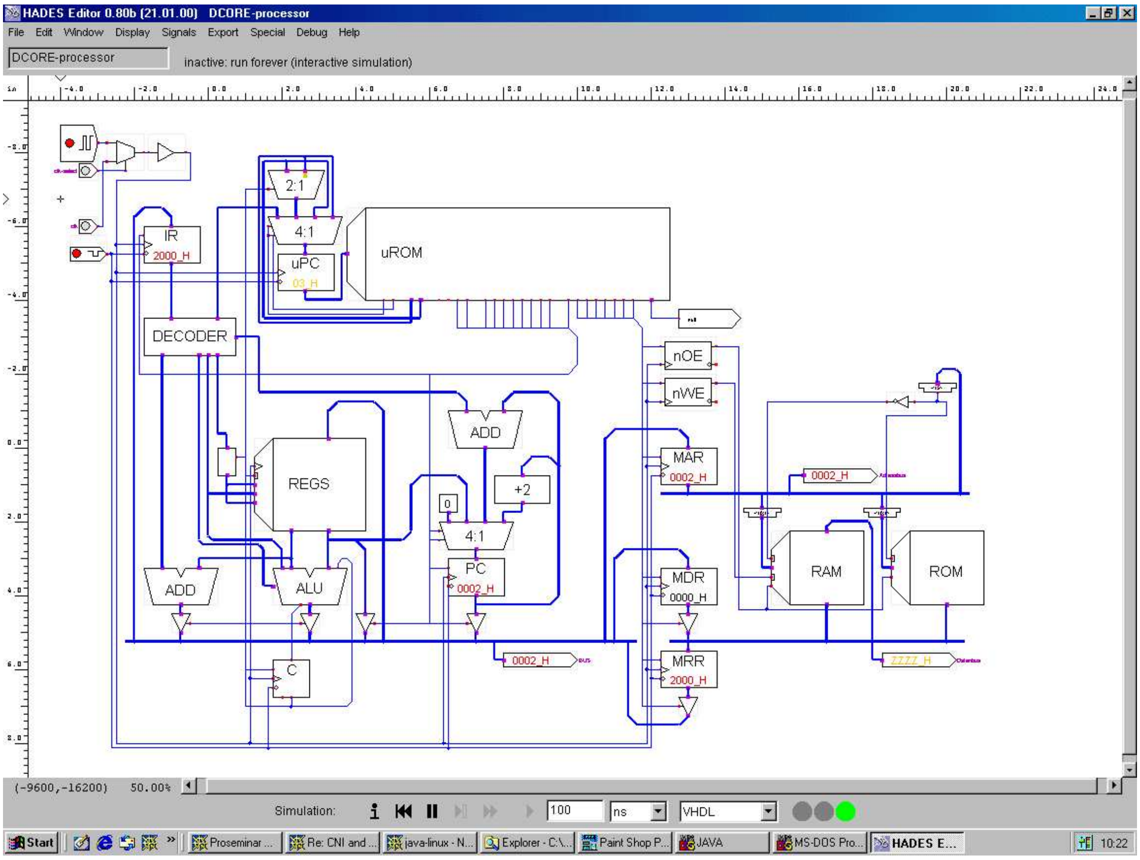

Abbildung 3.3: Das HADES D.CORE-Desgin stellt einen vollständigen Prozessor dar. Diese Abbildung zeigt die komplette HADES-Benutzeroberfläche. Dargestellt ist der HADES-Editor mit seinen Menüs, dem Design und am unteren Rand die Simulationssteuerung.

plexität und Baumtiefe ist dieses Vorgehen sehr arbeitsintensiv, bzw. unmöglich. Dieses Problem wird im Kapitel 8 auf Seite 57 aufgegriffen und im anschließenden Beispiel (s. Seite 61) elegant durch einfache Shellkommandos gelöst. Alle Designs werden zur Laufzeit zentral durch ein `DesignManager`-Exemplar verwaltet.

3.2.2 Abgrenzung zwischen logischer und graphischer Ebene

Das HADES-System läßt sich konzeptionell in zwei Ebenen aufteilen. Auf der logischen Ebene enstehen die Simulationen durch die Verbindung der Simulationsobjekte mittels Signalen. Deren graphische Repräsentation wird von der `Symbol`-Klasse übernommen, deren Exemplare durch den HADES-SymbolManager verwaltet werden. Der HADES-Editor stellt immer nur die Symbole der Simulationsobjekte des obersten Designs dar. Durch diese Aufteilung ist es für HADES möglich, komplexe Simulationen mit beachtlicher Performance auszuführen. Die graphische Darstellung ist besonders im Falle von Java sehr zeitaufwendig und daher zu minimieren. Auf der anderen Seite ist nur durch eine geeignete Darstellung eine interaktive, benutzerfreundliche Simulation zu realisieren. Die Trennung

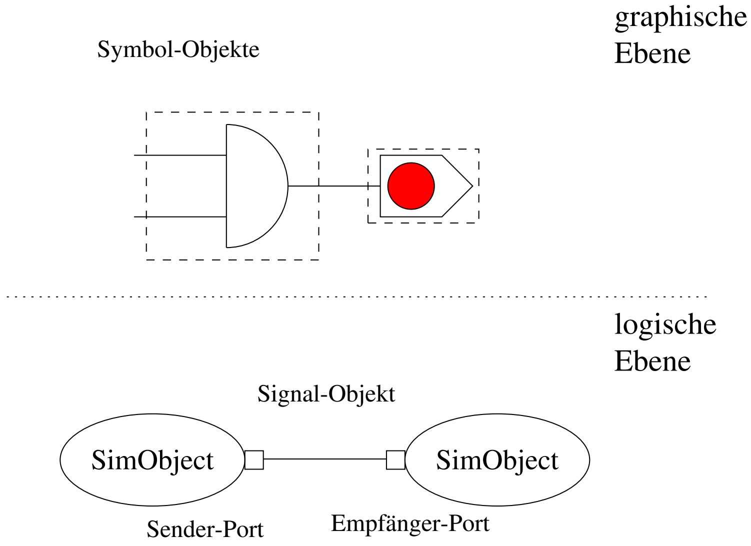

Abbildung 3.4: Das HADES-System trennt die logische Simulationsebene von der graphischen Darstellung der obersten Designkomponenten durch ihre Symbole. Der HADES-Editor stellt immer nur die Symbole der Simulationsobjekte des obersten Designs dar. Dieses läßt ihn bei tief verschachtelten Designs an Grenzen stoßen.

der logischen von der graphischen Ebene erfüllt beide Ansprüche und kann je nach Situation skaliert werden. Der HADES-Editor kann mittels eines *glow*-Modus Signalläufe sehr gut visualisieren, dafür sinkt die Anzahl der Ereignisse, die pro Sekunde verarbeitet werden können, drastisch. HADES-Simulationsexperimente, bei denen auf eine graphische Darstellung verzichtet wird, erreichen eine erstaunliche Performance. Für die Implementierung neuer Simulationskomponenten ist es wichtig zu wissen, daß eine neue Sub-Klasse von `SimObject` nur die logische Funktionalität spezifiziert. Deren graphische Repräsentation wird durch ein separates `Symbol`-Objekt realisiert, dieses liest eine HADES-Symboldatei (`.sym`) ein. In dieser Datei ist mittels der Klassen aus dem `hades.symbol`-Paketes das Aussehen der neuen Komponente festzulegen. Die beiden HADES-Klassen sind durch eine bidirektionale Referenzierung miteinander verbunden.

3.2.3 HADES-Schnittstelle für Simulationskomponenten

Jedes Simulationsobjekt eines Designs muß als Subklasse von `SimObject` realisiert werden und einige der dort definierten Methoden überschreiben. Durch das zu implementierende Java `Wakeable`-Interface erhält die Simulationskomponente die zusätzliche Fähigkeit, sich mittels Simulationsereignis vom HADES-SimKernel aufwecken zu lassen. Die folgende Übersicht beschreibt die grundlegenden Methoden der `SimObject`-Klasse. Diese werden von den Subklassen nach Bedarf überschrieben. Diese Grundlagen werden im Kapitel 10.2.1 auf Seite 78 wieder aufgegriffen und durch die Komponentenerweiterung der Skriptanbindung vertieft.

- Konstruktor:
  Hier muß als erste Methode `super()` (d.h. der Konstruktor vom `SimObject`) aufgerufen werden. Im Konstruktor werden alle Ein- und Ausgänge, die Verbindungen

zu HADES, definiert. Dazu existiert im `SimObject` ein Array mit Elementen der `Port`-Klasse und dem Namen `ports`. Für ein einfaches UND-Gatter sind dies z.B. die Eingänge `A` und `B` und der Ausgang `Y`.

- getSymbolResourceName():
  Diese Methode liefert den Namen des Symbols. Im Normalfall ist das der Klassenname mit der Dateiendung `.sym`.

- toString(): Der vollständige Name des Objektes wird von `toString()` geliefert.

- write():
  Soll ein HADES-Design gespeichert werden, hat jedes Modell die Möglichkeit, eigene spezielle Konfigurationsdaten in die zu schreibende Datei zu integrieren. Diese Konfigurationsdaten können dann bei erneuter Instantiierung von `initialize()` ausgewertet werden.

- initialize():
  Falls das Objekt eine spezielle Initialisierung benötigt, kann diese hier ausgeführt werden. Hier können zum Beispiel weitere Dateien für die Konfiguration des Objektes eingelesen werden. `initialize()` werden genau die Daten übergeben, die von `write()` erzeugt werden.

- elaborate():
  Für den Start der Simulation wird `elaborate()` aufgerufen. An dieser Stelle wird das Objekt für den Simulationsablauf selbst vorbereitet. Hier wäre der richtige Ort, initiale `Wakeup`-Ereignisse zu erzeugen (siehe Erklarung des `Wakeable`-Interfaces weiter unten) oder eine Referenz auf den Simulationskern zu speichern.

- evaluate():
  Wenn das Simulationsmodell ein Ereignis an einem der Ports des Modells feststellt, dann wird `evaluate()` aufgerufen. `evaluate()` muß dann alle Eingänge abfragen, auf mögliche Veränderungen reagieren und entsprechend interne Vorgänge anstoßen und eventuell neue Ereignisse an den Ausgangsports erzeugen. Zur Erzeugung von neuen Ereignissen existiert die Methode `getSimulator().scheduleEvent()`, die für einen bestimmten Port dessen nächsten Wert für einen bestimmten Zeitpunkt festlegen kann.

- configure():
  Falls es sich bei dem implementierten Modell um eine komplexere Anwendung handelt, die vom Benutzer konfiguriert werden kann, dann sollte das Modell die Methode `configure()` implementieren und bei dessen Aufruf dafür sorgen, daß eine Benutzerschnittstelle zur Konfiguration des Modells sichtbar wird.

Das `hades.simulator.Wakeable`-Interface definiert nur eine einzige zu implementierende Methode. Diese trägt den Namen `wakeup()` und wird aufgerufen, sobald ein Objekt aufgeweckt werden soll. Falls innerhalb eines zu simulierenden Bauteils interne zeitabhängige Vorgänge ablaufen, die nicht über die Eingänge gesteuert werden, muß diese Methode verwendet werden. Ein Beispiel für ein Bauteil, das mit dem `Wakeable`-Interface ausgestattet werden muß, ist ein Takterzeuger, der zum Generieren eines Taktsignals in definierten Zeitabständen aufgeweckt wird [Rink2000].

Teil II

Evaluation geeigneter Skriptsprachen und deren Interpreter

Kapitel 4

Aufbau der Untersuchung

Dieses Kapitel besteht aus zwei Themengebieten. Das erste verdeutlicht die Anforderungen an die Programmiersprachen und ihre Anbindungen an Java, um für diese Arbeit zum Einsatz zu kommen. Der zweite Teil stellt die geeigneten Skriptsprachen vor, deren Interpreter im Kapitel 5 auf Seite 29 evaluiert werden.

4.1 Anforderungen an die Skriptsprachen

Der folgende Abschnitt stellt verschiedene Skriptsprachen vor, von denen eine für die Realisierung dieser Arbeit ausgewählt werden soll.

Kriterien für die Auswahl einer Skriptsprache sind die Qualität der Sprache selbst, die Anbindung an Java, die Größe und Qualität ihrer Entwicklergemeinde. Diese ist es, die eine Sprache weiterentwickelt und eine schnelle und unbürokratische Hilfe gewährleistet. Die für diese Arbeit entscheidenden Faktoren sind eine möglichst einfache Anbindung an die bestehende Java-Applikation, die Möglichkeit zum interaktiven Testen, das Vorhandensein benutzbarer Module und Programme und nicht zuletzt die Plattformunabhängigkeit. Ein ebenfalls wichtiges Kriterium für die spätere Auswahl ist die bestehende Dokumentation, in Umfang und Qualität, sowie die schnelle Unterstützung durch die verantwortlichen Entwickler der Sprache. Anhand dieser Merkmale werden die Sprachen und deren Java-Interpreter im Kapitel 6 auf Seite 43 bewertet.

Zum Zeitpunkt dieser Arbeit konkurrieren die Skriptsprachen: Python, JavaScript [Fla97], Scheme und Tcl. VisualBasic als kommerzielles Produkt, das nur für eine Betriebssystem-Plattform (Microsoft Windows/NT) verfügbar ist, wurde nicht weiter beachtet. Auch für die verbreitete Skriptsprache Perl existiert mit JPerl [Bal99] eine Java-Anbindung. Diese befindet sich jedoch in einem unzureichend dokumentierten Betastadium, so daß dieser Ansatz für diese Arbeit nicht weiter verfolgt werden kann.

Eine besondere Stellung in dieser Untersuchung nimmt die Beanshell [Niem2000] ein. Hierbei handelt es sich nicht um die Anbindung einer bestehenden Skriptsprache an Java. Vielmehr basiert diese Skriptlösung komplett auf Java und zwar sowohl deren Entwicklung als auch deren Syntax. Die skriptspezifischen Anteile werden im Abschnitt 5.5 auf Seite 40 besprochen und der allgemeine Java-Anteil ist u.a. der angegebenen Literatur [GJS96] [Fla98] [Hen97b] [HC99] zu entnehmen.

Um eine bestehende Java-Applikation durch eine Skriptanbindung zu ergänzen, muß ein Interpreter in das System integriert werden. Dieser sollte vorzugsweise in derselben virtuellen Maschine wie die Java-Applikation laufen, oder auf eine andere Art Zugriff auf Java-Objekte zur Laufzeit realisieren. Ein Hinzufügen von Objekten muß ebenfalls unterstützt werden.

Der Interpreter sollte, unter dem Gesichtspunkt der bestehen Portabilität, in *100% pure-java* [Sun97] realisiert sein Sowohl der Interpreter als auch die Skriptsprache sollten frei verfügbar sein und wenn möglich im Quelltext vorliegen.

4.2 Vorstellung der Skriptsprachen

In diesem Abschnitt werden die Skriptsprachen vorgestellt, auf denen die entsprechenden Interpreter im Kapitel 5 auf Seite 29 aufsetzen.

4.2.1 EcmaScript

EcmaScript [ECMA99] basiert im wesentlichen auf der Sprache JavaScript 1.1 [Fla97] und dem Sprachkern von JScript [Jaw99], ohne die spezifischen Browser-Erweiterungen (z.B. von Netscape). Es ist eine objektorientierte Programmiersprache die explizit Objekte auf einem Hostsystem manipuliert. Dessen Funktionsweise wird durch eine genaue Schnittstellenspezifikation für das EcmaScript-Programm transparent gehalten. Entworfen wurde diese Sprache als Skriptsprache für das Internet. Sie kann aber auch als "normale" Skriptsprache benutzt werden. Es gibt die Unterstützung von einfacher Ein- und Ausgabe, Datei Ein- und Ausgabe, Zugriff auf Java-Objekte, Datenbank-Anbindungen und das auf regulären Ausdrücken basierende Suchen aus einem EcmaScript-Programm heraus. Ebenfalls beinhaltet es das dynamische Laden von Klassen und *JavaBeans* [Sun96]. Ein EcmaScript besteht aus mehreren kommunizierenden Objekten. In diesem Kontext wird unter einem Objekt eine Aufzählung von Eigenschaften und deren Attributen verstanden. Diese Eigenschaften können wiederum andere Objekte, primitive Werte oder Methoden enthalten. In EcmaScript gibt es keine Klassen, wie z.B. in Java. Konstruktoren werden zur Erzeugung von Objekten unterstützt, jede Funktion mit einem Konstruktor ist ein Objekt, aber nicht jedes Ecma-Objekt hat einen Konstruktor. In klassenbasierten, objektorientierten Programmiersprachen wird Zustand durch Exemplare dargestellt, Methoden werden in Klassen zusammengefaßt und nur Strukturen und Verhalten werden vererbt. EcmaScript verwendet ein anderes Konzept. Zustand und Methoden werden durch Objekte dargestellt und Verhalten, Struktur und Zustand sind vererbbar. In EcmaScript sind Typen eine Menge von Werten und primitive Werte, Exemplare der Typen Undefined, Null, Boolean, Number oder String.

4.2.2 Tcl

Tcl [Ous94] soll eine Programmierebene oberhalb von hardwarenahen und daher aufwendige Programmiersprachen wie C bieten. Tcl enthält fertige Routinen für Aufgaben, die beim Programmieren immer wiederkehren und den Programmierer sonst viel Zeit kosten würden. Die Sprache ist so konzipiert, daß die Skripte zur Laufzeit interpretiert werden. Die Idee zu Tcl entstand in den frühen 80er Jahren, als Prof. John K. Ousterhout an der University of California at Berkeley mit seinen Studenten an Programmen zum Entwurf von integrierten Schaltkreisen arbeitete. Durch das Fehlen einer universellen Allzwecksprache mußte für jedes Programm eine eigene Befehlssprache entwickelt werden, was zu einem nicht unerheblichen Aufwand führte.

Seine Vision basierte auf der Idee, Spezialteile weiterhin in Programmiersprachen wie C zu programmieren und dann mit Tcl-Routinen zu verkitten. Ein Tcl-Skript wird nicht kompiliert wie ein C-Programm, sondern durch den Tcl-Interpreter eingelesen und verarbeitet. Das Skript ist aus einzelnen Befehlen und Parametern aufgebaut und kann wie ein Shell-Skript in einer Datei stehen. Der Interpreter erkennt die Befehle und ruft passend dazu Routinen der Shell mit den entsprechenden Parametern auf.

Im Frühjahr 1988 begann Ousterhout bei der Firma Sun mit den Arbeiten, deren Ergebnisse zunächst Tcl und später TK (ein auf Tcl basierendes Toolkit für X11) waren und die heute von der Firma Scriptics weitergeführt werden.

Tcl ermöglicht es mittels der üblichen Programmierkonstrukte wie Variablen, Schleifen und Prozeduren, neue Applikationen zu schreiben und bestehende zu erweitern. Tatsächlich besteht der Interpreter aus einer einfachen C-Bibliothek, die zu bestehenden Applikationen einfach hinzugelinkt werden kann. Ein wichtiges Ziel von Tcl ist es, komplexe Anwendungen auf einem hohen Abstraktionsniveau entwickeln zu können, und im Vergleich zu C-Programmen unter Umständen ein Vielfaches an Quelltextlänge einzusparen und gegebenenfalls weniger Wissen um die interne Funktionsweise des zugrunde liegenden Systems zu benötigen.

In folgenden ausgewählten Bereichen eignen sich Tcl-Skripte hervorragend:

- Erstellung von Skripten für die Systemadministration
- Erstellung von benutzerfreundlichen Oberflächen zu vorhandenen benutzerunfreundlichen Programmen
- Entwicklung von kompletten, eigenständigen Anwendungen (Tcl-Webserver)
- Webskripting (CGI)
- Realisierung externer Hostanbindungen (z.B. 3270, Btx oder Cics)

Ferner kann man aufgrund der Tatsache, daß Tcl eine Interpretersprache ist, schnell, ohne großen Übersetzungsaufwand neue Anwendungen erstellen und ausprobieren (*rapid prototyping*). Es sei direkt angefügt, daß dies sicher auch zum größte Nachteil von Tcl geworden ist, da dieser Sprache nicht mehr zugetraut wird. Ein weiterer genereller Nachteil von in Interpretersprachen geschriebenen Applikationen ist, daß sie deutlich langsamer sind, als ihre maschinennahen übersetzten Versionen einer Systemsprache. Ousterhout versucht diese Einwände wie folgt zu entkräften: "Die heute verfügbaren Workstations sind unglaublich schnell. Ich habe Tcl-Skripte getestet, bei denen auf eine einzige Mausbewegung hin Hunderte von Anweisungen ausgeführt wurden, und konnte nicht die Andeutung einer Verzögerung ausmachen." [Ous94]

Neben diesen zentralen, zum größten Teil positiven Eigenschaften erhält der Entwickler einige weitere nützliche Eigenschaften, wenn er mit Tcl entwickelt. Die Sprache ist im Kern recht knapp gehalten, aber trotzdem mächtig, leistungsstark und bleibt ständig benutzerfreundlich. Sie verfügt über die Möglichkeit der Kommunikation mit beliebigen anderen Programmen (z.B *socket*-Kommunikation). Tcl-Skripte können zur Laufzeit ausgetauscht werden, was die Einbindung und das Exportieren von Programmteilen in andere

Programme sehr einfach macht und dementsprechend sicher ein wichtiger Faktor für die Dynamik von Tcl ist.

Eine weitere große Stärke von Tcl ist die leichte Erweiterbarkeit. So lassen sich neue Befehle ganz einfach in C schreiben, und können dann in Tcl eingebunden werden.

Neben all diesen Stärken und Eigenschaften von Tcl auf der technischen Seite ist sicher als überzeugendes Argument für Tcl auch dessen Verbreitung, seine extrem leichte Erlernbarkeit und die Entwicklungsfreude der Tcl-Benutzergemeinschaft zu nennen, denn gerade dadurch stehen regelmäßig viele neue Funktionen und Erweiterungen als Bibliotheken in Tcl zur Verfügung. Die ToolKit Erweiterung trägt einen großen Teil zum Erfolg von Tcl bei.

4.2.3 Scheme

Scheme ist ein kleiner, lexikalisch orientierter Lisp Dialekt mit exzellenten Möglichkeiten zur Abstraktion [DD96]. Die erste Version von Lisp wurde Ende der fünfziger Jahre von John McCarthy entwickelt und ist damit die älteste funktionale Programmiersprache. Lisp war auch die erste Sprache, die mit automatischer Speicherbereinigung (*garbage collection*) implementiert wurde. Scheme wurde 1975 von Guy Steele und Gerald Sussman ins Leben gerufen und stellt eine bereinigte Version von Lisp dar [ASS91]. Das ursprüngliche Lisp enthielt erstmals viele der Konzepte der funktionalen Programmierung, aber aus heutiger Sicht auch eine Reihe von schweren Designfehlern.

Scheme-Merkmale aus der LISP-Tradition sind:

- sprachliche Ausdruckskraft
- latente (dynamisch überprüfte) Datentypen
- interaktive Benutzerschnittstelle
- automatische Speicherverwaltung
- Repräsentation von Programmen durch Daten

Aus der ALGOL-Tradition bekam Scheme folgendes mit:

- Blockstruktur
- statische (lexikalische) Variablenbindung

Durch die geringe Größe von Scheme und seine einfache Semantik ist die Sprache einfach zu implementieren; mittlerweile existieren die meisten fortschrittlichen Erweiterungen, einschließlich Funktionen höherer Ordnung. Einer der Vorteile von Scheme liegt in der Einfachheit der Sprache. Alle Datenstrukturen bestehen nur aus Paaren oder Atomen und alle Funktionen haben die gleiche Notation wie die spracheigenen Kontrollstrukturen. Ein weiterer Vorteil, der vieles vereinfacht, ist die Konsequenz in der Syntax, die der Sprache zugrunde liegt. Egal ob es Schleifen, Variabledeklarationen, Zuweisungen, Funktionsaufrufe oder andere Sprachkonstrukte sind: alles hat in Scheme die Form:

<Klammer><Identifier><Argumente><Klammer zu>

Die einzelnen Token zwischen zwei Klammern werden nur durch Leerzeichen vneinander getrennt. Eine positive Konsequenz der Scheme Syntax ist das Ersparen von Semikolons zum Trennen von Funktionsaufrufen und Kommata bei Aufzählungen. Gewöhnungsbedürftig kann die in Scheme verwendetet Präfixnotation sein. Sie ist zwar bei Funktionsaufrufen vertraut, bei mathematischen Ausdrücken jedoch ungewöhnlich. Scheme wird immer beliebter, das liegt zum einen daran, daß interpretierte Sprachen immer größere Bedeutung erlangen, aber zum anderen auch daran, daß die *Free Software Foundation* (FSF) [FSF] einen eigenen Scheme-Interpreter entwickelt, der in Zukunft zentrale Skriptsprache in allen GNU-Programmen sein soll. Es handelt sich dabei um Guile [Har99], einen Scheme-Interpreter mit ein paar zusätzlichen Eigenschaften wie Ausnahmen, Prozesse etc. Die Verwendung von Scheme wird mitunter mißverstanden. Es geht nicht um die Programmierung mit Scheme, sondern um Wissen über Programmierung nach verschiedenen Grundmustern und Konzepten, das mit Scheme vorzüglich ausgedrückt werden kann. Scheme-Prozeduren beschreiben Prinzipien und Konzepte. Eine Sammlung von Scheme-Prozeduren bilden eine Arbeitsumgebung, dadurch werden verschiedene Programmierstile ermöglicht. Beispielsweise kann man sich eine solche für die logische bzw. für die objektorientierte Programmierung vorstellen. Arbeitsumgebungen können bei Bedarf nachgeladen werden. So hat man sehr schnell den gewünschten Kontext und kann das jeweilige Paradigma explorieren.

Die Hauptanwendungsgebiete von Scheme liegen in der Modellbildung, der symbolischen Informationsverarbeitung und der künstlichen Intelligenz.

4.2.4 Python

Python [Ros2000] ist eine leicht zu erlernende, mächtige und objektorientierte Programmiersprache, die ebenfalls wie Java auf hoher Ebene im *Bytecode* interpretiert wird. Ihre Entwickler haben sich bei ihrer Konzeption bewußt auf wenige Sprachkonzepte konzentriert und deren Erweiterbarkeit in den Vordergrund gestellt. Die Entwicklung begann 1989/1990 durch Guido van Rossum am CWI (Centrum voor Wiskunde en Informatica) in Amsterdam. Beeinflußt durch die Skripttechnologien, die durch die UNIX-Welt stark gefördert wurden und den Einfluß der objektorientierten Methoden und Konzepte entstand diese Sprache, die ihre Bewährung in der Betriebsystemforschung bestand. Obwohl Python rein objektorientiert entworfen und entwickelt wurde, unterstützt sie auch andere Programmierparadigmen wie die funktionale und die prozedurale Programmierung.

Python – benannt nach der britischen Comedy-Truppe "Monty Python" – zeichnet sich durch eine bemerkenswert klare und elegante Syntax aus. Ungewöhnlich ist die Blockbildung durch explizites Einrücken anstatt einer Klammerung. Python ist desweiteren durch einsatzstarke höhere Datentypen und moderne Sprachkonzepte sehr dynamisch und flexibel. Orthogonalität, klare Strukturierungsmittel und hohe Programmierkonzepte führen durch die vorgegebene Syntax zu klaren und kurzen Programmen, woraus eine deutliche Verkürzung des Entwicklungszyklus eines Projektes resultiert.

Der Python-Quelltext wird in Dateien mit der Endung "`.py`" gespeichert. Wird ein solches Skript (auch Modul genannt) von der Kommandozeile ausgeführt oder von einem Interpreter importiert, so wird der Quelltext durch Parsen und anschließendes Übersetzen in sogenannten *Bytecode* überführt und in einer Datei mit der Endung "`.pyc`" abgelegt. Dieser Bytecode wird dann vom Python-Interpreter ausgeführt.

Python besitzt sowohl ein Modulkonzept als auch die Möglichkeit mehrere Module in einem Paket zusammenzufassen. Die hierfür verwendete `import`-Anweisung wird in zwei Schritten ausgeführt. Zuerst wird das entsprechende Modul im Klassenpfad gesucht und wenn möglich initialisiert. Im zweiten Schritt wird ein Name für das importierte Modul im lokalen Namensraum (in dem die `import`-Anweisung aufgerufen wurde) angelegt. Klassen, Methoden und Funktionen sind in Python alles aufrufbare Objekte. Ihnen können verschiedenste Argumente beim Aufruf übergeben werden. Python unterstützt optionale Argumente, benannte Argumente sowie standardisierte Argumentlisten.

Da Sprachimplementationen für verschiedene Systemumgebungen (UNIX, Windows 3.x, Windows 97/98 und NT, OS/2 und Macintosh) existieren und diese, wie auch der Quelltext, frei verfügbar sowie einfach installierbar sind, ist auch der Einsatz der Skriptsprache in verschiedenen Anwendungsgebieten auf unterschiedlichen Plattformen möglich, womit zugleich ein hohes Maß an Portabilität der Software gewährleistet wird.

Python größte Stärke liegt in seiner Simplifizierung, deshalb wird Python oft als ausführbarer *pseudo code* bezeichnet [Ros98].

Kapitel 5

Evaluation der Interpreter

In diesem Kapitel werden die verschiedenen Interpreter vorgestellt. Sowohl deren Detaillierungsgrad als auch die Ausführlichkeit der einzelnen Besprechungen hängen stark von der vorhandenen Dokumentation ab. Dieses ist ein entscheidendes Kriterium für die spätere Auswahl eines Interpreters.

5.1 EcmaScript

5.1.1 Einführung in FESI

FESI (Free EcmaScript Interpreter) in der Release-Version 1.1.4 stellt eine komplette Implementierung von EcmaScript dar [Lug2000]. Zusätzliche Java-Pakete erlauben die Nutzung von EcmaScript als eine Skriptsprache für bestehende Java-Applikationen. Das Testen von Java-Bibliotheken, ist eine der zentralen Anwendungen für FESI. Ein interaktiver Interpreter ermöglicht das Ausführen und Testen von EcmaScript-Programmen. Die enthaltende Java-Bibliothek `FESI.jslib` ermöglicht es Java-Programmen, EcmaScript-Interpreter zu erzeugen, Skripte auszuführen und Eigenschaften von EcmaScript-Objekten zu lesen und zu schreiben.

5.1.2 Beschreibung der Implementierung von FESI

Der Zugriff auf Java wird über den FESI-Interpreter realisiert. Der Benutzer erhält direkten Zugriff auf alle Java-Klassen, die über den Klassenpfad erreichbar sind. Die Erweiterungen für den Java-Zugriff ist durch fünf Objekte aus dem `JavaAccess`-Paket realisiert, diese werden im folgenden dargestellt.

- Packages
  Ein `JavaPackages`-Objekt repräsentiert die Wurzel aller Java Pakete. Wird auf eine Eigenschaft eines `Packages`-Objekt zugegriffen, führt dies zur Erzeugung des entsprechenden `JavaPackages`-Objektes. Wird ein `Packages`-Objekt als Funktion eingesetzt, so kann hierdurch ein Java-“ClassLoader” erzeugt werden. Durch ihn können dann Klassen aus einem Verzeichnis oder einem jar-Archiv geladen werden.

- Beans
  Die Wurzel der Beanstruktur stellt das `JavaBeans`-Objekt dar. Hiernach können

JavaBeans relativ zu diesem Wurzelobjekt erzeugt werden. Wird das `JavaBeans`-Objekt als Funktion aufgerufen, können JavaBeans aus einem Verzeichnis oder Archiv heraus angelegt werden. Die Objektwerte der erzeugten Beans können durch den Eigenschaftenmechanismus von EcmaScript manipuliert werden.

- java
  Dieses Objekt ist eine Referenz auf das Basis Paket `Packages.java`, es vereinfacht den Zugriff auf die Funktionalität von Java.
- javaTypeOf
  Diese Funktion gibt den Klassennamen des erzeugten Java-Objektes zurück. EcmaScript-Objekte werden durch die `JSWrapper`-Klasse gekapselt, elementare EcmaScript-Werte werden in einen passenden Java Typ transformiert.
- loadExtension
  Diese Funktion lädt abhängig vom Klassennamen eine EcmaScript-Erweiterung nach.

Die folgenden kurzen FESI-Quelltextfragmente vermitteln einen Eindruck für das entstehende Sprachgemisch. Das erste Beispiel erzeugt ein Java-Fenster mit einem einfachen Java-Button und das zweite gibt die Java-Systemumgebung aus.

1. **FESI-Beispiel**

```
Frame  = java.awt.Frame;
JButton = Packages.javax.swing.JButton;

button = new JButton("Hello, world");
frm = new Frame();
frm.add(button, "South");
frm.pack();
frm.setVisible(true);

button.onAction="frame.dispose();exit();";
```

2. **FESI-Beispiel**

```
properties = java.lang.System.getProperties();

for (key in properties.keys()) {
  value = properties.get(key);
  writeln ("Property '" , key, " = '", value, "'");
}
```

Nachdem das `JavaAccess`-Paket geladen wurde, können Java-Objekte durch den `new`-Operator erzeugt und wie EcmaScript-Objekte angesprochen werden. Datentypen werden automatisch zwischen EcmaScript und Java konvertiert. Nicht kompatible Objekte werden durch ein Wrapper-Objekt gekapselt, das im weiteren die Zugriffe auf das enthaltende Java-Objekt realisiert. Wird ein solches Wrapper-Objekt an Java zurückgegeben, wird anstatt seiner Referenz die des gekapselten Java-Objektes zurückgeliefert. Wird der gemeinsame Zugriff von Java und EcmaScript auf ein Array benötigt, so muß ein Java-Array verwendet werden. Java Konstruktoren werden durch Funktionsaufrufe realisiert.

5.1.3 Bewertung

Durch die zugrundeliegende Sprache EcmaScript ist diese Java-Anbindung hauptsächlich auf den Einsatz im Internet ausgerichtet. Der zweite Schwerpunkt ist die Manipulation von JavaBeans. Diese Schwerpunkte decken sich nicht mit den Anwendungszielen dieser Arbeit. Die bestehende Dokumentation von FESI ist lückenhaft und teilweise unzureichend. Durch den Java Einfluß von JavaScript und die Unterstützung einiger OO-Eigenschaften wirkt die Interaktion mit Java recht natürlich, wobei elementare Sprachkonstrukte wie z.B. die Ausnahmebehandlung fehlen.

5.2 Tcl

5.2.1 Einführung in Jacl

Auf der JavaOne Konferenz 1997 stellte Sun eine Skripttechnologie für Java vor, Jacl (*Java Command Language*) und Blend [Joh98]. Jacl ist ein in 100% pure java implementierter Tcl-Interpreter. Sowohl Jacl als auch die gesamte Skriptsprache Tcl befindet sich im Besitz der Firma Scriptics. Bei dem Tcl-Blend Projekt handelt es sich um die neue Möglichkeit, einen normalen Tcl-Interpreter durch Java anstatt C zu erweitern. Für die Realisierung dieser Arbeit ist eine genauere Betrachtung von Tcl-Blend nicht sinnvoll. Jacl wurde komplett in Java entwickelt und erweitert die Plattformunabhängigkeit von Tcl um die Plattformen der Java Stations und Betriebssysteme, für die es eine virtuelle Maschine für Java, aber keine Tcl Portierung existiert. Tcl hat sich als Skriptsprache für C eine beeindruckende Fangemeinde, geführt von John K. Ousterhout, geschaffen. Mit Jacl will Tcl nun seinen Siegeszug fortsetzen. Portierungen werden durch die Ausnutzung der Plattformunabhängigkeit von Java vermieden. Tcl und auch Jacl sind frei verfügbar, dieses betrifft ebenfalls den entsprechenden Quelltext.

5.2.2 Beschreibung der Implementierung von Jacl

Im Gegensatz zu den anderen Java-Anbindungen führt Jacl den Java Quelltext sofort aus, es findet kein Übersetzungsschritt statt. Jacl ermöglicht, über die Java-Reflection-API auf Java-Objekte zuzugreifen. Realisiert wird dies durch ein zusätzliches Java-Paket. Eines der Hauptziele von Tcl war und ist es, bestehende C-Applikationen durch das einfache Einfügen des Tcl-Interpreters um eine Skriptanbindung zu erweitern. Dieses Designziel wird ebenfalls von Jacl verfolgt, zusätzlich wird die Plattformunabhängigkeit von Tcl erweitert. Jacl, in der Version 1.2, basiert auf dem Java-Developer-Kit (JDK) 1.1 von Sun, einige der neuen Eigenschaften des JDK 1.2 werden ebenfalls unterstützt, z.B der Unicode-Zeichensatz. Jacl erbt den Sprachumfang von Tcl 8.2 und wird analog weiterentwickelt. Verbessert wurden der Import-Mechanismus und die Behandlund von Ausnahmen . Gänzlich neu ist die Einführung des Namensraumkonzeptes von Java in Jacl. Anhand des folgenden Beispiels wird auf die syntaktischen Unterschiede zwischen Java und Tcl eingegangen.

Java-Quelltext

```
import java.awt.Button;
import java.awt.Frame;

public class TestAWT {
```

```
    public static void main(String[] argv) {
      Frame f  = new Frame();
      Button b = new Button("Press Me");

      f.setSize(300,300);
      f.add(b);
      f.show();
    }
}
```

Jacl-Quelltext

```
package require java

java::import java.awt.Button
java::import java.awt.Frame

set f [java::new Frame]
set b [java::new Button "Press Me"]

$f setSize 300 300
$f add $b
$f show
```

Schon durch dieses kleine Beispiel werden die elementaren Unterschiede der beiden Sprachen deutlich. Die Jacl-Variante ist, durch das fehlen der Klassendeklaration, kürzer und hat den Vorteil, daß es nicht übersetzt werden muß. Es wird direkt vom Interpreter ausgeführt und kann interaktiv angegeben werden. Das Jacl-Skript entspricht einem normalen Tcl-Skript, die einzige Änderung ist die neue Abhängigkeit von der Java Erweiterung und deren Kommandos.

Tcl ist nicht objektorientiert und verfügt daher weder über ein Klassenkonzept noch über eigentliche Objekte. Sie werden durch einfache Variablen modelliert. Die Java-Erweiterungen sind durch einen eigenen Tcl-Namensraum gekapselt. Dieser konzeptionelle Unterschied wird bei der Erzeugung von Java-Objekten in Jacl deutlich. Es wird aus dem `java`-Namensraum die `new`-Funktion aufgerufen, ihr erstes Argument ist die Signatur des anzusprechenden Konstruktors, die folgenden seine Argumente. Der zurückgelieferte Wert wird einfach einer Tcl-Variablen zugewiesen. Durch die oberflächliche Untypisiertheit von Tcl sind diese immer als String anzusehen.

```
set s [java::new String "I am a Java String"]
```

Die allgemeingültige Tcl-Syntax besteht immer aus der Abfolge:

<Kommando><Name><Argumente>

dies behält innerhalb von Jacl seine Gültigkeit. So werden Java-Methodenaufrufe, ebenfalls durch reine Funktionsaufrufe auf den Inhalt der entsprechenden Tcl-Variablen realisiert.

```
$s charAt 7
>> J
```

Durch das `tcltest`-Paket können schnell und sehr effizient Testumgebungen für Java implementiert werden. Das nachfolgende Quelltextfragment stellt eine Prüfroutine für die `charAt()`-Methode von Strings dar. Das erste Argument des `test`-Befehls ist der Testname, gefolgt von einer Beschreibung und dem zu testeten Quelltextblock, das letzte Argument nimmt das erwartete Ergebnis auf.

```
test charAt { make sure the java.lang.String.charAt() method works} {
set s [java::new String "I am a Java String"]
$s charAt 7
} J
```

Die geschweiften Klammern führen ausschließlich nur zur Entwertung der Leerzeichen innerhalb der Argumente. Leerzeichen sind in Tcl das Trennzeichen der Syntax.

Dieser Testmechanismus kann besonders hilfreich bei der Entwicklung von *Enterprise-Java-Beans (EJB)* [Sun2000] [MH2000] sein. Deren Verhalten kann durch geeignete Tests spezifiziert werden und dieselben Tests dienen später zur Überprüfung der Implementierung.

5.2.3 Bewertung

Mit Jacl steht für die Skriptsprache Tcl eine mächtige Java-Anbindung zur Verfügung. Die entstandene Sprach-Synthese ist etwas gewöhnungsbedürftig, hat aber ihre speziellen Reize. Für erfahrende Tcl-Entwickler stellt Jacl eine sehr reizvolle Alternative dar, um sich Java zu nähern. Der syntaktische und konzeptionelle Bruch mit der Objektorientierung wird für Java-Entwickler nur schwer verständlich und annehmbar sein. Jacl bleibt reines Tcl, erweitert um den Namensraum `java::` und bietet somit gutes Potential für eine angeregte Diskussion unterschiedlichster Auffassungen der Softwareentwicklung. Seine Vorteile liegen auf der Hand, der Entwicklungszyklus ist gewohnt schnell und die enorme Dynamik von Tcl bleibt erhalten. Dem gegenüber steht eine starke Verfremdung von Java. Die Dokumentation ist ausführlich und exakt, der gebotene Support ist schnell und unbürokratisch. Jacl ist geeignet für pragmatische Ansätze, z.B das Testen von Java-Applikationen. Bestehende Tcl-Skripte können bei einer Migration einer C-Applikation zu Java weitgehend beibehalten werden.

5.3 Scheme

5.3.1 Einführung in Skij

Skij [Tra98] stellt eine interaktive Skriptsprache zur schnellen Erstellung von Prototypen in einer Java-Umgebung dar. Skij ist ein kleiner Scheme Interpreter, der in Java implementiert und in der normalen virtuellen Maschine von Java läuft. Er beinhaltet Erweiterungen, die es erlauben, interaktiv Java Objekte zu erzeugen, zu untersuchen und zu verändern. Der Skij-Interpreter kann leicht in bestehende Applikationen eingebunden werden und eignet sich als Java Programm und Objekt Debugger-Interface.

5.3.2 Beschreibung der Implementierung von Skij

Das wichtigste Design Prinzip bei der Entstehung von Skij war, daß Skij-Objekte und Java-Objekte gleichwertig sind. Jedes Java-Objekt kann in ein Skij-Objekt umgewandelt werden und jedes Skij-Objekt in ein Java-Objekt. Skij benutzt möglichst die eingebauten Java-Klassen, dies führt manchmal zu einer Divergenz bezüglich des Scheme Standards. Alle numerischen Werte werden in der typischen Scheme-Box durch die Benutzung der Java `Integer`-Klasse und `Double`-Klasse realisiert. Scheme-Strings finden ihre Entsprechung in den Java-Strings.

Die Java-Anbindung von Skij basiert im wesentlichen auf den folgenden Funktionen.

- (new class args*)
  erzeugt ein neues Java-Objekt, wobei der mit den Argumenten übereinstimmender Klassen-Konstruktor benutzt wird (s. Quelltextbeispiel, Zeile 2).
- (invoke object method-name args*)
  die spezifizierte Methode wird auf das übergebene Objekt angewendet und deren Ergebnis zurückgeliefert (s. Zeile 3).
- (invoke-static class method-name args*)
  realisiert den Zugriff auf Java-Klassenmethoden.
- (peek object field-name)
  gibt den Wert des angegebenen Feldes zurück (s. Zeile 8). Der Zugriff auf statische Klassen-Felder ist analog zum Aufruf von Methoden realisiert.
- (poke object field-name new-value)
  verändert den Wert des angegebenen Feldes, eine statische Version existiert ebenfalls.

Das folgende interaktive Quelltextbeispiel zeigt, wie auf Java durch Skij zugegriffen werden kann und wie sich die Scheme-Syntax auswirkt. In der Skij-Shell wird ein einfacher Java-Dialog aufgebaut und angezeigt.

```
1 skij> (invoke button 'setLabel "Please Press Me")
2 skij> (define window (new 'java.awt.Frame))
3 skij> (invoke window 'setVisible #t)
4 skij> (invoke window 'add button)
5 skij> (invoke window 'getSize)
6 skij> (invoke window 'setSize 300 200)
7 skij> (invoke window 'show)
8 skij> (peek (invoke button 'getSize) 'width)
```

Skij profitiert und erhält durch Java eine automatische Speicherverwaltung, numerische Typen, Strings und Arrays. Umgebungsvariablen werden durch die Verwendung von Java-Hashtables auf höchster Skij-Ebene realisiert. Der *scheme reader* ist als ein Java `StreamTokenizer`-Objekt implementiert. Andere Sprachkonzepte wie z.B. Paare, variierende Prozedur-Typen und Ports wurden direkt in Skij konzipiert und umgesetzt.

Ein Sonderfall stellt der Java `null`-Wert dar, er kann auf einen korrekten Skij-Wert abgebildet werden, aber leider nicht generell. Eine Ausnahme ergibt sich dadurch, daß der Wert einer Variablen auf der obersten Ebene kein `null`-Wert sein darf. Funktionen die z.B. das oberste Element eines Stapels liefern, können in ihrer statischen Form einen `null`-Wert zurück liefern, diese müssen durch eine Prüfung erkannt und in einen anderen Wert verändert werden.

5.3.3 Bewertung

Skij kann durchaus als Skriptsprache für Java in Betracht gezogen werden, da alle Standard-Charakteristiken einer Skriptsprache vorhanden sind. Auf die vorhandenen Java-Schnittstellen kann zugegriffen werden und die eigentliche Sprache ist mächtiger als die typischen Skriptsprachen. Es gibt noch andere Java-Implementierungen von Scheme, wobei einige wesentlich komplexer sind, als Beispiel sei hier Kawa [Bot98] erwähnt. Kawa übersetzt Scheme-Quelltext direkt in Java-Bytecode und führt diesen schneller als Skij aus. Kawa ist wesentlich größer und Scheme-lastiger. In dieser Arbeit liegt der Schwerpunkt in der dynamischen, interaktiven Skriptsteuerung einer Java-Applikation. Hier hat Skij, als Scheme ähnliche Skriptsprache, für Java seine Vorteile. Negativ ist die gänzlich andere Syntax von Scheme, ein gemischter Quelltext entfremdet Java stark.

5.4 Python

5.4.1 Einführung in JPython

Python wurde von Grund auf objektorientiert entworfen und entwickelt und paßt deshalb sehr gut zu Java. Aber der Hauptgrund, warum Python für den Einsatz in dieser Arbeit als Skriptsprache für Java zu betrachten ist, ist die Existenz von JPython [Hug97b]. Aufgrund der erstaunlichen Entwicklung von Java in den vergangenen zwei Jahren ist Java auf dem Weg, einen neuen Standard zu setzen. Auch John Hugunin ist diese Entwicklung nicht verborgen geblieben. Im April 1997 begann er am MIT mit der Entwicklung von JPython, einen Python Interpreter, der in 100% pure java geschrieben ist.

Die Java-Implementierung (JPython) hat im Vergleich zur C-Implementierung (CPython) einige Vorteile, wie z.B. Portabilität, Stabilität und Objektorientierung. Nachteilig ist die Tatsache, daß Java wiederum eine interpretierte Sprache ist. JPython wird interpretiert von einem Java-Interpreter, der seinerseits auch nur interpretiert wird. Dieses Performance Problem könnte mit neueren und besseren *just-in-time-compilern (JIT)* und Java-Virtual-Machinen für Java bald in den Hintergrund treten. Messungen von John Hugunin haben ergeben, daß JPython (mit Python 1.5) etwa 25% bis 30% der Leistung von CPython erbringt [HM99]. Neuere Messungen mit der Hotspot-VM von Sun (Version 2) [Sun99] zeigen, daß nach einigen Durchläufen die Ausführungsgeschwindigkeit von JPython nahezu identisch mit der von CPython ist!

Die direkte Übersetzung des Python-Bytecodes in den Java-Bytecode ist der Grund für die enorme Geschwindigkeit von JPython. Die anderen Skriptsprachen sind im Durschnitt wesentlich langsamer (s. Abschnitt 6.1.4 auf Seite 44). Die Integration von Python und Java durch JPython geschieht äußerst nahtlos. Dies wird durch das *Introspektion Feature* der Java-Reflection-API ermöglicht. Introspektion erlaubt es, zur Laufzeit das Interface einer Klasse zu ermitteln. Somit ist es anderen Objekten z.B. möglich, zur Laufzeit herauszufinden, welche Methoden andere Objekte zur Verfügung stellen.

Ein JPython-Skript kann alle Java-Klassen importieren, die über den Python-Klassenpfad erreichbar sind. Python kann statische- und Exemplar-Methoden von Java-

Klassen ausführen, Exemplare erzeugen und Subklassen von Java-Klassen bilden. Exemplare dieser Subklassen können an Java zurückgegeben werden und deren Methoden (in Python geschrieben) von Java aus aufgerufen werden.

Als Beispiel ein JPython "Hello World" Applet:

```
from java.applet import Applet

class HelloWorld(Applet):

  def paint(self, gc):
      gc.drawString('Hello World', 20, 30)
```

Dieses Programmbeispiel skizziert einige der wichtigen Eigenschaften von JPython: Es läuft in jedem Browser (der das JDK 1.1 unterstützt). Es wird eine Java-Klasse importiert und eine Subklasse davon erzeugt, die `paint`-Methode wird von der Java-AWT-Ereignisroutine aus aufgerufen.

5.4.2 Beschreibung der Implementierung von JPython

Die Implementierung von Python in Java basiert im Wesentlichen auf drei Komponenten. Die erste ist ein Parser und Lexer für die Python-Grammatik, dieser Java-Code wurde von Sun's JavaCC Parsergenerator erzeugt. Der Parser generiert aus Python-Quelltext Java-Objekte, die den *parser-tree* darstellen. Die zweite Komponente des JPython-Systems ist ein in Python geschriebener Übersetzer, der den *parser-tree* in Java-Bytecode übersetzt. Dies ist ein erstes Beispiel für die Vorteile dieses Systems, der Python-Übersetzer bearbeitet eine Baumstruktur aus Java-Objekten und erzeugt Java-Bytecode. Dies ist ein entscheidender Unterschied zur C-Implementierung, die ein eigenes Bytecode-Format erzeugt. Die dritte Komponente der Python-Implementierung in Java stellt die Sammlung von Java-Klassen für die Python-Basis-Objekte dar.

Übersetzung von Python, in Java-Bytecode

Sowohl Python als auch Java werden in einen portablen Bytecode übersetzt. Die virtuellen Maschinen (VM) der beiden Sprachen sind beide Stack-basiert, unterscheiden sich aber in einem Punkt wesentlich. Die VM von Python wurde entworfen, um Python-Bytecode auszuführen und bei der Entwicklung der Java-VM wurde darauf geachtet, daß sie auch Bytecode anderer Sprachen ausführen kann. Es gibt einige Übersetzer-Projekte für z.B. Scheme, Tcl, Python, Ada und C, die auf die Java-VM aufsetzen [Tol2000].

Das Problem, wie der Python- in den Java-Bytecode übersetzt wird, soll einleitend anhand eines sehr einfachen Bytecode-Beispiels veranschaulicht werden.

Als erstes wird der Python-Bytecode des Python-Übersetzers für die Anweisung "2+2" betrachtet:

Python Bytecode für 2+2

```
0  LOAD_CONST  0  (2)
3  LOAD_CONST  0  (2)
6  BINARY_ADD
```

Es wurden drei Zeilen Python-Bytecode generiert. Die ersten beiden Ausdrücke laden ein Python Integer-Objekt in den Operandenstack. Die letzte Zeile addiert die beiden obersten Elemente des Operandenstacks.

Zum zweiten folgt der Java Bytecode für den selben Ausdruck:

Java Bytecode für 2+2

```
0  iconst_2
1  iconst_2
2  iadd
```

Auch bei Java ergeben sich drei Zeilen Bytecode. Die ersten beiden legen einen Integer “2” auf den Operandenstack ab und die die letzte Zeile addiert die beiden obersten Operanden.

Die beiden Bytecode-Beispiele sehen fast identisch aus, aber sie unterscheiden sich erheblich in einem Punkt. Die Java-VM arbeitet mit Integer-Werten im 32-Bit Binärformat; der Wert 0x00000002 repräsentiert den Integer 2. Im Gegensatz hierzu manipuliert die Python-VM Integer-Objekte. Es werden keine Integer-Werte auf dem Stack abgelegt, sondern Zeiger auf ein Integer-Objekt. Darüber hinaus sind Python’s `BINARY_ADD`- und Java’s `iadd`-Anweisung sehr unterschiedlich. Die Python-Anweisung ruft dynamisch den passenden Programmcode auf, um die beiden obersten Objekte des Stacks zu addieren. Dies geschieht abhängig von deren zur Laufzeit erkannten Typen. Die Java Bytecode `iadd`-Anweisung setzt voraus, daß die beiden obersten Operanden zwei 32-Bit Integers sind. Dieses einfache Beispiel zeigt deutlich den Unterschied der beiden virtuellen Maschinen, die Python-VM ist auf hohe Interaktivität und Dynamik ausgelegt, während die Java-VM auf höchstmögliche Performance optimiert wurde.

Trotz dieses Unterschiedes hat die Java-VM die Fähigkeit, das dynamische Modell der Python-VM mit geringem Overhead zu implementieren. Das erste Teilproblem, daß Java Integer keine geeignete Repräsentation für Python’s Integer-Objekte darstellen, wird durch die Verwendung von Java Objekten als Repräsentation gelöst. Die fehlende Dynamik der Java `iadd`-Anweisung wird einfach durch einen Java Methodenaufruf erfüllt. Durch diese Änderungen kann nun die Python-Semantik für “2+2” in Java-Bytecode übersetzt werden.

Java Bytecode für 2+2 mit P2 mit Python Semantik

```
0 getstatic #1     //static field _i2
3 getstatic #1     //static field _i2
5 invokevirtual #2 //PyObject.add(LPyObject;)LPyObject;
```

Diese Implementierung legt ein Java-Objekt für das korrespondierende Python Integer-Objekt “2” auf dem Stack ab, dies geschieht zweimal. Die dritte Anweisung ruft auf dem obersten Objekt seine `add`-Methode auf, deren Implementierung die Python “add” Semantik beinhaltet.

Für alle in Python enthaltenen Typen mußte in JPython eine Java-Klasse implementiert werden (s. Abb. 5.1). Daraus resultiert die folgende Java-Klassenhierarchie der Python-Typen, die für eine einfache automatische Typenkonvertierung benutzt werden.

Die Namen der Java-Klassen sind denen aus Python sehr ähnlich. Der unschöne “Py”-Prefix resultiert leider aus der C-Implementierung von Python. Java besitzt zwar ein besseres Namensraum-Management durch seinen Paket-Mechanismus, aber durch den automatischen Import des `java.lang`-Paketes jedes Java Programmes sind die Namen wie

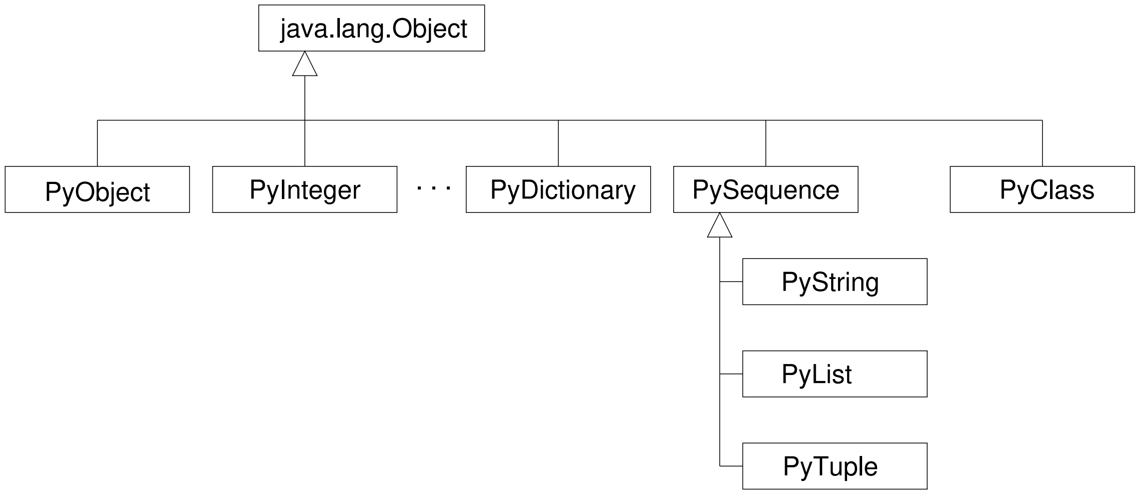

Abbildung 5.1: Für jeden Python-Typ existiert in JPython eine Java-Klasse, deren Basisklasse die java.lang.Object-Klasse ist. Die Python-Typen String, List und Tuple erben von ihrer eigene Basisklasse, diese ist wiederum eine Subklasse von der java.lang.Object-Klasse.

"Object", "Class" oder "Integer" bereits reserviert. Die `PyObject`-Klasse beinhaltet die gesamte Basisfunktionalität für die dynamischen Methodenaufrufe von Python.

Einen Sonderfall stellen Strings dar. Python-Strings sind im Gegensatz zu Java-Strings keine Objekte und haben keine Methoden. Ein Methodenaufruf führt in der Regel zu folgendem Fehlerfall:

```
>>> 'hello'.indexOf("e")
Traceback (innermost last):
  File "<console>", line 1, in ?
 AttributeError: 'string' object has no attribute 'indexOf'
```

Auf die Arbeitsweise der Übersetzung des Bytecodes von Python zu Java wird nun ausführlicher eingegangen. Das Hauptaugenmerk wird hierbei auf die konzeptionelle Vorgehensweise gelegt, dadurch können die angeführten Beispiele in Form von Java bzw. JPython-Quelltext angegeben werden. Dieses erhöht die Lesbarkeit und das Verständnis gegenüber Bytecode.

Zunächst wird der Python-Quelltext für das Modul `foo.py` dargestellt und anschließend die generierte Java-Version ausführlich diskutiert.

```
x = 2
print x+2

import spam
spam.eggs = 2
print spam.eggs
```

Der Java-Quelltext zum korrespondierenden Java-Bytecode, der von J. Hugunins Übersetzer (*jpythonc* JPython Version 1.1) generiert wurde: `foo.java`

```
1   public class foo implements PyRunnable {
2     //Deklaration der Konstanten
3     static PyInteger _i2 = new PyInteger(2);
4     static PyString _sx = new PyString( "x" );
5     static PyString _sspam = new PyString( "spam" );
6     static PyString _seggs = new PyString( "eggs" );

7     public void run(PyFrame frame) {
8       //x = 2
9       frame.setlocal(_sx, _i2);

10      //print x+2
11      Py.print(frame.getlocal(_sx).add(_2), true);

12      //import spam
13      frame.setlocal(_sspam, Py.importModule(_sspam));

14      //spam.eggs = 2
15      frame.getlocal(_sspam).__setatrr__(_seggs, _2);

16      //print spam.eggs
17      Py.print(frame.getlocal(_sspam).__getattr__(_seggs));
18    }
19  }
```

Zeile 1 deklariert eine neue Java-Klasse. Für jedes Python-Modul wird genau eine Java-Klasse erzeugt. Diese muß das Interface `PyRunnable` implementieren, um das Modul initialisieren zu können.

Zeile 3-6 definiert die benötigten Konstanten in Form von statischen Java-Klassenfeldern. Jedes dieser Klassenfelder enthält ein Java-Objekt das mit den primitiven Python-Konstanten korrespondiert.

Zeile 7-18 definiert die `run`-Methode des Interfaces der Java-Klasse. Diese wird beim Importieren des Python-Moduls aufgerufen. Der Methode wird ein `PyFrame`-Objekt übergeben, das mit dem Python-Stack, der das Module ausführt, übereinstimmt. In diesem Objekt sind alle lokalen und globalen Pythonvariablen enthalten.

Zeile 9 illustriert die Benutzung des `Frame`-Objektes bei einer einfachen Variablenzuweisung. Der lokalen Variable, dessen Name mit dem Python String “x” übereinstimmt, wird die Python-Integer Konstante 2 zugewiesen.

Zeile 11 veranschaulicht ein interessanteres Beispiel. Zum Wert der lokalen Variablen x wird mittels der `add`-Methode die Python Integer-Konstante 2 addiert. Die Implementierung dieser `add`-Methode befindet sich in der `PyObjekt`-Klasse und realisiert Python's dynamischen Methodenaufruf. Ebenfalls zu beachtet ist, daß der Java-VM's Ausführungsstack benutzt wird, um die Operanden dieser `add`-Methode zu verwalten. Weiterführend wird die automatische Speicherverwaltung der Java-VM benutzt, so daß kein Zählen der Objektreferenzen mehr benötigt wird.

Zeile 13 ist eine einfache `import`-Anweisung, deren gesamte Funktionalität in der

`Py`-Klasse zu finden ist. Aus Komplexitätsgründen wird auf die Realisierung der `importModule`-Methode hier nicht näher eingegangen. Sie wird hier als eine komfortable Möglichkeit benutzt, um ein Objekt den lokalen Namensraum zuzufügen. Hiernach können die Objekt-Attribute, mittels der lokalen Referenz manipuliert werden.

Zeile 15 demonstriert das Setzen eines Objektattributes. Die `__setattr__`-Methode wird auf einem Java-Objekt angewendet und verhält sich identisch zu ihrer Python-Variante. Das Java-Objekt ist wiederum repräsentiert als ein `PyInstance`-Objekt.

Zeile 17 gibt das eben manipulierte Objektattribut aus. Die `__getattr__`-Methode ist äquivalent zu ihrer Python-Version.

JPython-Programme können als Java-Bytecode weitergegeben werden, so wird ein erneutes Kompilieren nicht notwendig und der Programmcode ist auch gegen *reverse engineering* weitgehend geschützt.

5.4.3 Bewertung

JPython führt die beiden objektorientierten Sprachen Java und Python homogen zusammen. Objekte können analog zu Java angelegt und verwendet werden, zusätzlich können die dynamischen Skriptsprachenanteile der eleganten Python-Syntax genutzt werden. JPython stellt eine Java-Reimplementation von Python dar, deren Ausführungsgeschwindigkeit enorm ist. Die Dokumentation ist umfassend und die Aktivitäten des Entwicklungsteams sind kontinuierlich gewachsen. Ein umfassendere Bewertung von JPython befindet sich im Abschnitt 6.2 auf Seite 45.

5.5 Beanshell

Die Beanshell [Niem2000] in der Version 1.0 von Pat Niemeyer ist ein kleiner, frei verfügbarer und einbindbarer Quelltext-Interpreter für Java, geschrieben in Java. Ihren Namen erhielt sie aus der Anlehnung an die "bash" (the GNU "Bourne again" shell) und ihrer Unterstützung für die Entwicklung von JavaBeans. Der Grund für die Entstehung war das Fehlen einer kleinen Skriptanbindung für Java; eine Shell für Java fehlte Pat Niemeyer für seine Projekte. Die derzeit schon bestehenden Lösungen wie Jacl oder JPython waren zu groß und für seine Zwecke wurde ihr mächtiger Funktionsumfang kaum genutzt. Die Beanshell kann nicht nur die Standard Java-Ausdrücke und Anweisungen ausführen, sondern zusätzlich enthält dieser Interpreter Skriptbefehle und Skriptsyntax. Hierbei gemeint sind seine Möglichkeiten, die statische und strenge Typisierung von Java aufzuweichen bzw. aufzuheben. Der Skriptanteil der Beanshell unterstützt die Erzeugung von Objekten und das einfache Deklarieren von Methoden. Die einfache Skriptanbindung der Beanshell an Java eignet sich für das interaktive Testen und Debuggen von Java-Applikationen, vorzugsweise von JavaBeans. Mit ihr verfügt der Anwender über ein dynamisch interpretiertes Java, mit allen Vorteilen einer Skriptsprache. Der Interpreter der Beanshell läßt sich einfach in bestehende Applikationen einbetten, aber das Hauptgewicht dieser Lösung liegt in der graphisch gestalteten Benutzer-Shell.

Die Java-Ausdrücke und Anweisungen werden durch die Java-Reflection-API [McM97] zur Laufzeit interpretiert, die hieraus resultierenden Geschwindigkeitseinbußen sind für den interaktiven Einsatz zu vernachlässigen. Aufgrund der vorteilhaften Eigenschaft, direkt Java-Quelltext interpretieren zu können, erhält die Beanshell alle Vorteile von Java und ergänzt sie sinnvoll, um benötigte Skriptsprachen-Konzepte wie z.B dynamisch erzeugte untypisierte Variablen.

5.5.1 Einführung in die Beanshell

In diesem Abschnitt wird auf die Spracheigenschaften der Beanshell näher eingegangen. Durch kleine Quelltext-Beispiele sollen die einzelnen Punkte verdeutlicht werden.

Innerhalb der Beanshell kommen verschiedene Syntax Formen zum Einsatz. Außer der Standard Java-Syntax stehen auch noch eine aufgeweichte Java-Syntax sowie die vereinfachte Syntax zur Verfügung.

- Standard Java-Syntax:

```
JButton a = new JButton("a Standard Button");
```

- Aufgeweichte Java-Syntax:

```
b = new JButton("a loosy Button");
```

- Vereinfachte Syntax:

```
c = new java.awt.Button();
c{"label"} = "a convenience button";
// Äquivalent zu: c.setLabel("a convenience button");
```

Im Gegensatz zur statischen und strengen Typisierung der Standard Java-Syntax benutzt die aufgeweichte Syntax eine dynamische Typisierung. Im obigen Beispiel wurde der b-Button vorher nicht deklariert. Der Typ dieser Zuweisung wird zur Laufzeit ermittelt und dem Objekt zugewiesen. Die vereinfachte Syntax kommt bei der Arbeit mit JavaBeans zum Einsatz, um deren Attribute schnell und einfach zu setzten.

Methoden können innerhalb der Beanshell genau wie in Java deklariert werden. Es besteht ebenfalls die Möglichkeit die dynamische Typisierung für die Argumente und den Rückgabewert zu nutzen. Im zweiten Fall "kümmert" sich die Beanshell um die korrekte Typenkonvertierung.

- Java-Syntax

```
int addTwoNumbers( int a, int b ) {
    return a + b;
}
```

- Dynamische Variante

```
add( a, b ) {
    return a + b;
}

foo = add(1, 2);
```

```
print( foo ); // >> 3

foo = add("Hello", "World");
print( foo ); // >> Hello World
```

Außer den Java-Methoden enthält die Beanshell auch noch eigene nützliche Skriptbefehle, die die Skripteigenschaften unterstützen sollen. Enthalten sind Befehle für das Arbeiten mit Dateien (cd, dir, load, rm usw.), Prozesse (bg, run) und Skriptsprachen-Kommandos wie z.B source, eval und exec [Niem2000b].

Das Namensraumkonzept von Java wurde um eine `global`-Anweisung erweitert, mit der man Objekte auf oberster Skriptebene bekannt machen kann.

Skriptobjekte werden ähnlich wie in EcmaScript (s. Abschnitt 4.2.1 auf Seite 24) in Form von Funktionsaufrufen angelegt.

Die Beanshell ist in der Lage, als Exemplar des `event handling`-Interface von Java zu agieren, ebenfalls können Threads in ihr benutzt werden.

Ein Satz von speziellen Variablen und Werten runden den Funktionsumfang der Beanshell ab. Ein sogenannter `Server`-Mode befindet sich im Betastadium, durch den verteilte Zugriffe realisiert werden sollen. Die Beanshell kann einfach um eigene Kommandos erweitert werden, hierzu sind lediglich geeignete `jar`-Archive in einem entsprechenden Verzeichnis zu hinterlegen.

5.5.2 Beschreibung der Implementierung der Beanshell

Zu der internen Struktur der Beanshell existiert keine geeignete Dokumentation. Die Beanshell basiert hauptsächlich auf der Java-Reflection-API. Um die Grammatik innerhalb der Beanshell umzusetzen, wurde JavaCC (Version 1.1) von Sun eingesetzt. Der Beanshell-Parser ist aufgrund seiner Implementierung (Abhängigkeit von vielen kleinen Klassen) bei den ersten Aufrufen merkbar langsam.

5.5.3 Bewertung

Der Anspruch der Beanshell ist es, neben den mächtigeren Java-Anbindungen von Jacl und JPython als Alternative für die Entwicklung und Testen von JavaBeans zu gelten. Sie ist bewußt sehr klein und kompakt gehalten. Die Beanshell bietet die Möglichkeit Java als reine Skriptsprache einzusetzen und unterstützt dies durch geeignete Skriptsprachen-Kommandos. Ihr Vorteil gegenüber anderen Skriptanbindungen ist, daß keine zweite Sprache erlernt werden muß. Für größere Projekte ist die Beanshell nicht konzipiert, sie ist eine gute und effiziente Java-Shell.

Kapitel 6

Bewertung der Java-Anbindungen

Dieses Kapitel faßt die gewonnen Erkenntnisse aus der Evaluation der vorhandenen Skriptsprachen und ihrer Interpreter zusammen. Begründet wird die Auswahl von `JPython` für den praktischen Teil dieser Arbeit.

6.1 Bewertung der Untersuchung

Jeder der betrachteten Interpreter ist in 100% `pure java` implementiert und ermöglicht eine Interaktion mit bestehenden Java-Objekten. Sie sind alle wie ihre Basissprachen frei verfügbar. Keine der untersuchten Skriptsprachen bringt von sich aus die Möglichkeit einer Java-Anbindung mit. Durch ihre erweiterten Interpreter, lassen sich neue Java-Objekte erzeugen und Methoden auf Objekten oder Klassen aufrufen. In den folgenden Abschnitten werden die untersuchten Interpreter und deren Sprachen verglichen. Eine Zusammenfassung beendet diese Evaluation.

6.1.1 Spracheigenschaften

Durch eine Anbindung einer Skriptsprache an Java ensteht eine Mischform dieser beiden Sprachen. Für deren Einsatz sind zwei wichtige Punkte zu beachten. Zum einen, welcher Syntax die gemischten Java-Skripte unterliegen und zweitens, welche Spracheigenschaften Java hinzugewinnt. Bei den untersuchten Sprachen differiert dies erheblich, eine Bewertung ist deshalb stark von der Anwendungsdomäne des Gesamtsystems abhängig. Für diese Arbeit sind hierbei keine signifikanten Vor- oder Nachteile festzustellen.

Die Syntax der Skripte entspricht der der Skriptsprache. Für die Bewertung ist die Ähnlichkeit mit der Syntax von Java entscheidend, dadurch "fühlt" sich eine Skriptanbindung sehr positiv an. Gemischter Java-Skriptsprachen-Quelltext bleibt für Java-Kenner leicht verständlich und vertraut, die Eingewöhnungzeit sinkt.

Die beste Syntax der untersuchten Skriptsprachen bietet eindeutig Python. Deren klare Syntax ist eines der Hauptpunkte für den Erfolg dieser Sprache. Sie ist der Syntax von Java nicht nur mehr als ähnlich, sie ist dieser sogar vorzuziehen. JPython-Quelltext ist kürzer und übersichtlicher als reine Java-Varianten. Die anderen untersuchten Skriptsprachen sind im Punkte ihrer Ähnlichkeit mit Java weit abgeschlagen. Nur die Beanshell mit ihrer aufgeweichten Java-Syntax kann in diesem Punkt ebenfalls positiv abschneiden. Dabei ist keine neue Sprache zu erlernen, wodurch auch keine neuen Eigenschaften hinzukommen.

6.1.2 Konzept der Interpreter

Für die technische Bewertung der Interpreter ist die Realisierung des Zugriffs auf Java ein wichtiges Kriterium. Hierbei konkurrieren verschiedene Ansätze. Die Beanshell greift ausschließlich über die Reflection-API auf Java zu. Während Jacl den Quelltext direkt interpretiert, setzt JPython auf die Übersetzung in Java-Bytecode. Auf eine Mischform aus direktem Übersetzen und Verwendung der Java-Reflection-API setzen Skij und FESI. Diese unterschiedlichen Konzepte resultieren aus den Anwendungsgebieten der Interpreter. Um Java-Bibliotheken oder JavaBeans zu testen, ist die Funktionalität der Java-Reflection-API ausreichend, für intensivere Projekte aber zu langsam. Bei den beiden Mischform-Ansätzen, dominieren die Skriptsprachen sehr stark und Java nimmt die Rolle einer neuen Schnittstelle ein.

Die Konzepte von Jacl und JPython setzen ihre Basissprache komplett in Java um. Für sie stellt Java eine neue Plattform dar und sie wollen als Skriptsprache für Java gelten. Die fehlende Objektorientiertheit von Tcl schränkt deren Einsatzgebiet stark ein.

Die Beanshell wurde in diesem Punkt unter Vorbehalt bewertet, über deren Internas existierte keine geeignete Dokumentation.

6.1.3 Dokumentation und Support

Für die Bewertung der Dokumentation ist deren Umfang, Qualität und Aktualität wichtig. Hierbei ist zwischen der Skriptsprache und der Dokumentation des Interpreters zu unterscheiden. Unter Support wird die Unterstützung durch die Entwickler- und Fangemeinde der verschiedenen Projekte verstanden.

Negativ aufgefallen ist in diesem Punkt der Skij-Interpreter, seine Dokumentation ist sehr knapp gehalten und stellt sein Ko-Kriterium dar. Alle anderen Projekte sind ausreichend dokumentiert. Bei Jacl ist eine lange Pause von über einem Jahr im Support festzustellen. Erst in den letzten Monaten wurde dieses Projekt wieder aufgenommen. Die EmcaSkript-Sprache von FESI ist wesentlich ausführlicher dokumentiert als der Interpreter.

Positiv herauszustellen ist JPython, die Dokumentation ist umfassend und geht tief in die technischen Details. Die für diese Arbeit erhaltene Unterstützung seitens des JPython-Supports war schnell und effizient.

6.1.4 Geschwindigkeit

Für diese Arbeit ist die Ausführungsgeschwindigkeit nicht von zentraler Bedeutung. Aufgrund der gewählten Konzepte sind in diesem Punkt nur Jacl und JPython zu vergleichen, die Benutzung der Java-Reflection-API ist dagegen sehr langsam.

Wird die reine Ausführungszeit für ein einfaches Skript gemessen, so ist Jacl sehr schnell. Der Grund hierfür liegt darin, daß Jacl das Skript nicht erst in Bytecode übersetzt, sondern es direkt interpretiert. Dieser Vorteil von Jacl schwindet jedoch bei anderen Szenarien. Wird ein Skript mehrmals ausgeführt und ist es von etwas höherer Komplexität, gewinnt JPython mit seiner Übersetzung in Java-Bytecode deutlich an Geschwindigkeit. Wenn das JPython-Skript einmal übersetzt wurde, ist dessen Ausführung sogar schneller als die Interpretation von Jacl.

Für einfache nicht objektorientierte Skripte, die nicht schnell hintereinander ausgeführt werden, stellt Jacl deshalb eine sehr schnelle Lösung dar. Für sich mehrfach wiederholende,

	Sprache	Interpreter	Dokumentation	Geschwindigkeit
FESI	–	–	++/(+)	k.A.
Jacl	– –	+	+	++
Skij	–	–	– –	k.A.
JPython	+++	+++	++++	+++
Beanshell	+	(+)	– –	– –

Tabelle 6.1: Entscheidungs-Matrix der Untersuchung

komplexere Aufgaben ist JPython vorzuziehen, besonders wenn es um die reine Ausführung von vorübersetzten Bytecodes geht.

6.1.5 Zusammenfassung

Die Ergebnisse der Untersuchung werden in der Tabelle 6.1 zusammengefaßt. Resümierend ist zu sagen, daß jede untersuchte Skriptlösung einen in sich stimmigen Eindruck macht. Diese unterscheiden sich stark in ihren Anwendungsgebieten und setzen deshalb deutlich andere Schwerpunkte. Der FESI-Interpreter ist durch die zugrunde liegende JavaScript-ähnliche Sprache sehr Internet-lastig. Das interaktive Testen von EcmaScript-Programmen und JavaBeans ist das Betätigungsfeld von FESI. Nachteilig sind die Größe und die unzureichende Dokumentation des Interpreters.

Mit Jacl steht ein guter und schneller Interpreter für Tcl zur Verfügung. Die Skriptsprache Tcl ist schnell zu erlernen und ab der Version 8.0 sehr mächtig und kompakt. Der entscheidende Nachteil dieser Lösung ist das komplette Fehlen objektorientierter Konzepte. Die Syntax von Tcl harmoniert nicht mit der von Java und beide Sprachen verfügen mit TK und Swing über eigene Bibliotheken für graphische Benutzeroberflächen.

Der Scheme-Interpreter Skij ermöglicht eine Verbindung von Java mit seiner sehr mächtigen Basissprache. Die aus der funktionalen Programmierung stammende Syntax verträgt sich nur schwer mit der von Java. Leider ist die bestehende Dokumentation des Interpreters mangelhaft und ließ nur eine sehr oberflächliche Untersuchung zu.

JPython stellt Java die elegante Syntax von Python zur Verfügung. Für Python ist dieser Interpreter eine weitere Sprachimplementierung, die sich durchaus mit der C-Implementierung messen kann. JPython ist sehr gut dokumentiert und ist im Durchschnitt deutlich die schnellste Skriptanbindung für Java. Die Fähigkeiten von JPython übertreffen die der einzelnen Sprachen Python und Java. JPython eignet sich für größere Projekte, die Sprachkombination ist sehr mächtig und elegant. Ein sehr interessanten Ansatz stellt die Beanshell dar. Sie besteht nur aus Java und keiner zweiten Sprache. Der wichtigste konzeptionelle Entschluß ist ihre "Größe", die Beanshell ist mit ca. 135 KByte der kleinste Interperpreter. Er wurde zum Testen und Debuggen von JavaBeans konzipiert. Für größere Projekte ist die Beanshell aber nicht vorgesehen.

6.2 Begründung der Auswahl von JPython

Dieser Abschnitt begründet die Auswahl von JPython für die Implementierung einer Skriptanbindung für das HADES-Framework. Hierbei werden die Vorteile von Python betont, sowie die Gemeinsamkeiten mit Java betrachtet.

6.2.1 Der erste Eindruck und die Entscheidung

JPython besticht schon beim ersten Eindruck durch Klarheit, Verständlichkeit und Eleganz. Keine Syntax der anderen Skriptsprachen harmoniert so mühelos mit den Java-Anweisungen wie die von Python. Beide Sprachen sind voll objektorientiert konzipiert und entwickelt. Sie unterstützen dieselben OO-Paradigmen und basieren beide auf der Interpretation ihres Bytecodes. JPython-Quelltext ist für einen Java-Kenner ebenso mühelos zu lesen, wie für einen der nur Python kennt. Das verwendete technische Konzept, das Übersetzen des JPython-Quelltextes in Java-Bytecode, ist robust und enorm schnell. Die gute und ausführliche Dokumentation bildet die Grundlage für die Entscheidung, JPython in der Realisierung zu verwenden. Die meisten der evaluierten Skriptsprachen sind speziell für ein Anwendungsgebiet zugeschnitten, die sich nicht mit dem dieser Arbeit decken.

6.2.2 Vergleich der evaluierten Skriptsprachen und Python

Der anschließende Vergleich von Python mit den anderen untersuchten Skriptsprachen stellt deren Gemeinsamkeiten und Unterschiede dar.

JavaScript

Der objektorientierte Sprachkern von Python und JavaScript ist im weitesten Sinne äquivalent. Sowohl JavaScript als auch Python unterstützen (anders als Java) auch den nicht objektorientierten Programmierstil. Es können einfache Funktionen und Variablen ohne umständliche Klassendefinitionen verwendet werden. Der Funktionsumfang von JavaScript ist hiermit abgeschlossen, im Gegensatz hierzu bietet Python für die Entwicklung größerer Applikationen noch deutlich mehr. Die vollständige Unterstützung der objektorientierten Programmierparadigmen erhöhen die Wiederverwendbarkeit von Python-Objekten, echte Vererbung und Klassen sind hier ein wichtiger Faktor.

Tcl

Tcl und Python verbindet, daß beide Skriptsprachen für die Erweiterung und Einbettung in bestehende Applikation konzipiert wurden. Weitere Gemeinsamkeiten sind kaum festzustellen. Tcl 8.2 hat aber durch das Hinzufügen vieler wichtiger Sprachkonzepte, z.B. Namensräume, zu Python aufgeschlossen. Das komplette Fehlen von objektorientierten Ansätzen in Tcl ist der Hauptunterschied zu Python. Ein weiterer ist die wesentlich schwächere Typisierung. Insbesondere der erste Punkt schränkt Tcl im Vergleich zu Python deutlich in den geeigneten Anwendungsgebieten ein. Die strenge, klare Syntax von Python steht im Gegensatz zur kreativen Freiheit von Tcl.

Lisp und Scheme

Diese Sprachen sind sich in ihrer Dynamik sehr ähnlich, so können sie alle Programme zur Laufzeit erzeugen und ausführen. Ihre komplett verschiedenen Syntax-Konzepte lassen einen objektiven Vergleich kaum gelingen. So sind Fragen wie; "Ist das Fehlen von Syntax in Lisp ein Vor- oder Nachteil ?" nicht eindeutig zu beantworten und werden seit langer Zeit intensiv diskutiert. Unterschiede liegen aber im Alltag dieser Sprachen. Common-Lisp ist in jedem Sinne riesig, von Scheme existieren weltweit mehrere inkompatible Versionen, und Python existiert hierzu im Gegensatz in einer kompakten und freien Implementierung.

6.2.3 Abschließende Betrachtung von Java und Python

Die Vor- und Nachteile der Sprachen Java und Python werden in der folgenden Betrachtung verglichen und deren Gemeinsamkeiten unterstrichen.

Objektorientiert (kurz OO)

Java ist ausschließlich eine objektorientierte Sprache. Prozedurale Programmierung außerhalb von Klassendefinitionen wird nicht unterstützt. Dies zwingt den Entwickler dazu, sich auf Objekt-Hierarchien zu konzentrieren; Prozeduren werden zu Methoden, die auf einem Objekt angewendet werden. Diese Denkweise hat sich für mittlere bis große Projekte bewährt, hier bringt die Objektorientierung Vorteile. Kleine Applikationen werden aber unverhältnismäßig zerstückelt und durch eine starre OO-Entwicklung überkonzeptiert. Python ist von Grund auf objektorientiert konzipiert, aber auch funktionales und prozedurales Programmieren wird unterstützt, und somit stehen mehrere Paradigmen gleichberechtigt nebeneinander und können für die geeigneten Probleme herangezogen werden. Sowohl Java als auch Python benutzen die qualifizierende Punktnotation (z.B. objekt.attribut), so daß auch ein gemischter JPython-Quelltext nichts von der Übersichtlichkeit von Python verliert.

Aufgrund der starken Typisierung von Java muß der Typ aller Variablen explizit deklariert sein, bevor sie initialisiert werden können. Ebenso muß die Anzahl der Argumente und ihr Typ zur Übersetzungszeit festgelegt sein, es gibt keine variablen Argumentenlisten. Im Gegensatz hierzu besitzt Python eine dynamische Typisierung und ein statisches Namensraumkonzept. Variablen erhalten bei ihrer Instantiierung den Typ des Objektes, der sich aus der rechten Seite der Zuweisung ergibt. Variable Argumentenlisten, Standard- und benannte Argumente sind im Python-Sprachumfang enthalten.

Beide Sprachen unterstützen das späte Binden von Objekten zur Laufzeit. In Java wird zuerst in der aktuellen Klasse nach dem Kontext eines Objektes gesucht und danach die Vererbungskette bis zur `java.lang.Objekt`-Klasse emporgestiegen. Durch die Unterstützung der Mehrfachvererbung durch Python ergibt sich hier ein Vererbungsbaum, der nach dem *depth-first, left-to-right*-Prinzip durchsucht wird.

Während in Java alle Objekte immer von einem Basis-Objekt erben, sind in Python alle Entitäten sogenannte *first-class*-Objekte.

Module

Sowohl Java als auch Python haben einen `import`-Mechanismus, der es ihnen erlaubt, zur Laufzeit Gruppen von ausführbaren Programmcode hinzuzunehmen. In Python ist ein Modul eine Gruppe von Klassen oder Funktionen, die sich alle in einer Quelltextdatei befinden. Mehrere Module können in Python weiter zu Paketen gruppiert werden, die sich hierarchisch auf Verzeichnisse verteilen. In Java befindet sich jede Klasse in einer eigenen Datei, und mehrere Klassen können zu einem Paket innerhalb eines Verzeichnisses zusammengefaßt werden.

Typensystem

Obwohl Python viele Eigenschaften mit Java teilt, wie die Interpretation von Bytecode und die Unterstützung des späten Bindens von Objekttypen, ist sein Typensystem sehr unterschiedlich. Python ist im Gegensatz zu Java eine dynamisch typisierte Sprache, Fehler durch nicht zusammenpassende Typen können erst zur Laufzeit erkannt werden. Wie Java, so hat auch Python einen mächtigen Ausnahmebehandlungs-Mechanismus, der es erlaubt, Laufzeitfehler sehr elegant zu bewältigen.

Behandlung von Laufzeitfehlern

Für Sprachen, die ein spätes Binden zur Laufzeit unterstützen, ist ein guter Mechanismus zur Ausnahmebehandlung unerläßlich. Java und Python verfügen über fast dieselben Kontrollstrukturen und Methoden. Ein Unterschied ist, daß in Python Fehler typischerweise immer String-Objekte sind, während sie in Java immer Objekte des *throwable-Interfaces* sind.

Teil III

Realisierung der HADES-Skriptanbindung

Kapitel 7

Konzeption der Skriptanbindung

Dieses Kapitel geht, nach der Vorstellung der grundlegenden Architekturidee einer Skriptanbindung, konkret auf die Architektur der HADES-Skriptanbindung ein. Hierbei liegt der Schwerpunkt auf der Vermittlung des Konzeptes, die Funktionalität der einzelnen Komponenten wird in den anschließenden Kapiteln vorgestellt.

7.1 Grundlegende Architektur einer Skriptanbindung

Um eine bestehende Applikation um eine Skriptanbindung zu erweitern, wurde die in Abbildung 7.1 dargestellte Architektur gewählt. Ihre Grundidee sowie die daraus resultierenden Vorteile sollen im Folgenden dargestellt werden.

Die Grundarchitektur gliedert sich grob in vier Bereiche: Applikation, Interpreter, Basisskripte und Benutzerskripte. Die Aufgabe besteht nun darin, die ausgewählte Applikationslogik einer Skriptsprache zugänglich zu machen. Die eigentliche Applikation soll im Kern nicht verändert werden und wird lediglich um einen Interpreter erweitert. Im Idealfall erzeugt die Applikation ein Interpreter-Objekt und übergibt ihm Referenzen auf die zu manipulierenden Applikationsobjekte. Dieser Interpreter beinhaltet keinerlei Anwendungslogik, er initialisiert die benötigte Laufzeitumgebung indem er z.B. seine Ein- und Ausgaben in eine Applikations-Oberfläche umleitet. Innerhalb dieses Interpreters werden nach der Initialisierung die sogenannten Basisskripte ausgeführt. Sie kapseln die Applikationslogik und stellen die Abstraktionsschnittstelle für die Benutzerskripte dar. Diese Basisskripte stellen das Grundsystem einer Skriptanbindung dar und sollten nicht von den Benutzern verändert werden. Für ein vom Benutzer erstelltes Skript ist die Applikation transparent und besteht aus elementaren Anweisungen der Skriptsprache, sowie Methoden und Prozeduren aus den Basisskripten. Diese strikte Aufteilung ermöglicht es, die Funktionalität für die Benutzer genau zu kontrollieren und die Benutzerskripte effizient und kurz zu halten. Ein direkter Zugriff auf Objekte und Methoden der Applikation erhöht die Flexibilität der Skriptanbindung, eine Benutzerkontrolle ist aber nicht mehr möglich. Durch diese Architektur sind die Schnittstellen zwischen der Applikation und den Benutzerskripten für beide Seiten ideal. Die Schnittstelle ist aus der Sicht der Applikation minimal, da sie nur um ein Interpreter-Objekt erweitert wurde. Änderungen innerhalb der Applikation stehen der Skriptanbindung unmittelbar zur Verfügung, der Interpreter verwaltet nur die Objektreferenzen. Wird z.B. ein Applikationsobjekt um eine Methode erweitert, muß die Skriptanbindung nicht angepaßt werden und die neue Methode steht sofort zur Verfügung.

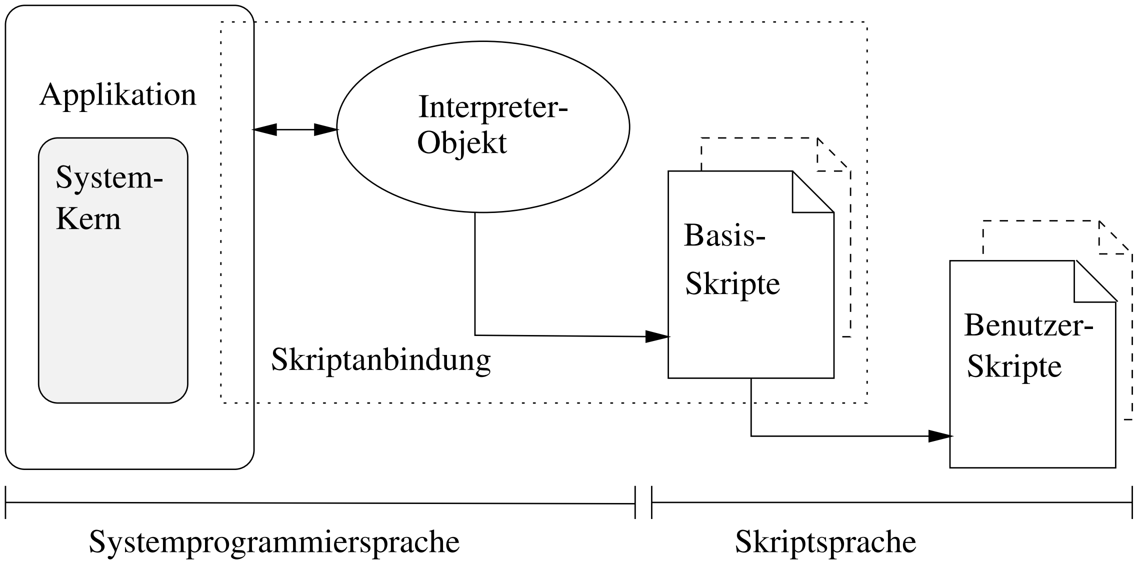

Abbildung 7.1: Die grundlegende Architektur einer Skriptanbindung sollte die Applikation nur minimal verändern, der Systemkern bleibt unberührt. Die Skriptanbindung umfaßt den Interpreter und die Basis-Skripte, sie stellen den Benutzer-Skripten die Applikationslogik zur Verfügung.

Aus der Sicht des Benutzers stellen die Basisskripte eine eindeutige Schnittstelle dar, die ihm helfen, sich auf die Realisierung seiner Skripte zu konzentrieren.

7.2 Architektur der HADES-Skriptanbindung

In diesem Abschnitt wird basierend auf der Grundarchitektur das Konzept der entwickelten HADES-Skriptanbindung vorgestellt. Sie unterteilt sich in zwei funktional unterschiedliche Bereiche. Zum einem beinhaltet sie die Möglichkeit, daß der Benutzer mittels einer Shell interaktiv mit dem HADES-System interagieren kann. Diese sogenannte interaktive Skriptsteuerung (`jpshell`) stellt die Basis der Skriptanbindung dar. Deren Architektur wird im folgenden Abschnitt vorgestellt, ihr Funktionsumfang im Kapitel 8 auf Seite 57. Anschließend an die Gesamtstruktur der HADES-Skriptanbindung wird das grobe Konzept der Komponentenerweiterung von HADES vorgestellt. Auf deren Realisierung geht das Kapitel 10 auf Seite 75 vertiefend ein. Die Komponentenerweiterung durch Python-Module basiert auf der Funktionalität der `jpshell` und kann mittels der HADES-STL getestet werden.

7.2.1 Gesamtstruktur der HADES-Skriptanbindung

Die HADES-Skriptanbindung gliedert sich anlehnend an die Grundarchitektur in zwei Teile, die miteinander interagieren. Der erste Teil besteht aus den Klassen des Java-Paket `hades.jpshell` und den zweiten bilden die JPython-Basismodule (s. Abb. 7.2).

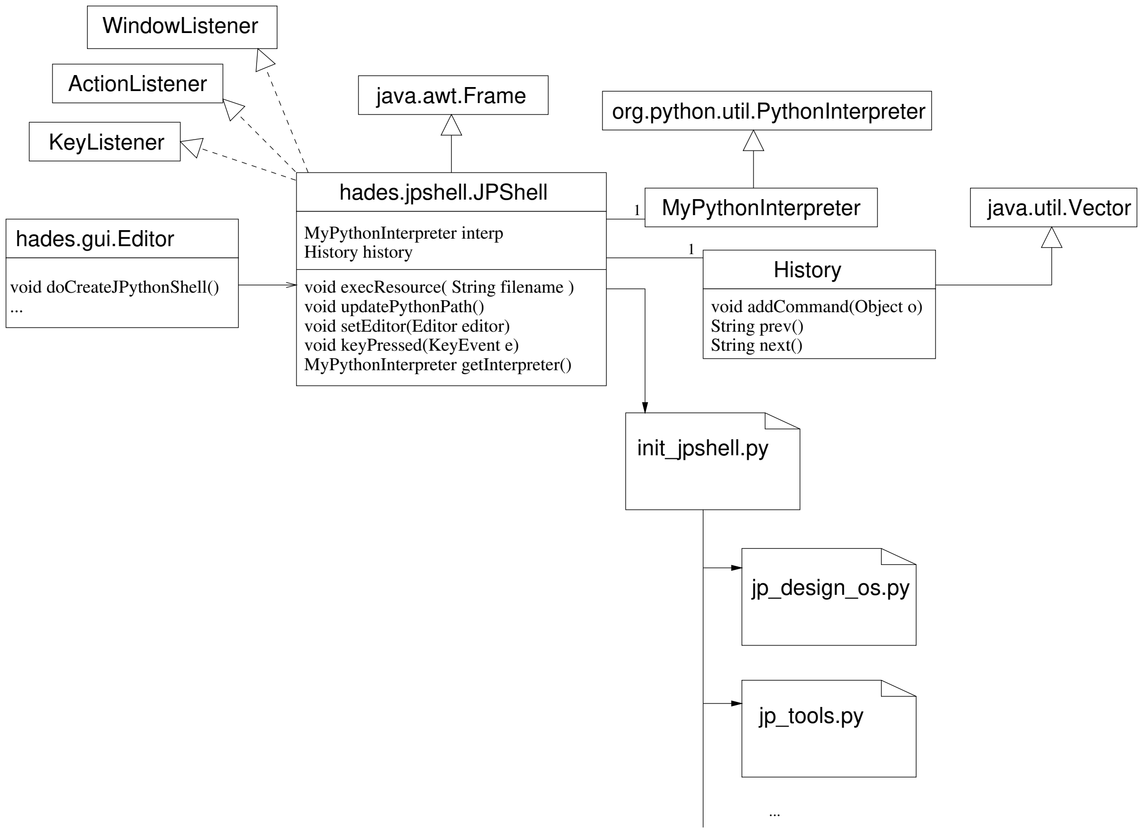

Abbildung 7.2: Die JPShell-Klasse nimmt die zentrale Rolle in der Klassenstruktur der HADES-Skriptanbindung ein. Sie enthält das Exemplar des Python-Interpreters und erstellt das GUI der Shell. Der HADES-Editor erzeugt die JPShell-Objekte, diese initiieren den Interpreter und laden das Python init_jpshell-Modul.

Die zentrale Rolle im Java-Teil übernimmt die `JPShell`-Klasse. Ein Exemplar dieser Klasse wird aus dem HADES-System heraus erzeugt. Sie stellt die graphische Oberfläche für die Benutzerinteraktion zur Verfügung und enthält das Exemplar der `MyJPythonInterpreter`-Klasse. Dieser erweitert den normalen Python-Interpreter, um die Möglichkeit der Umlenkung seiner Ausgabe und stellt die Ausführungsumgebung für die JPython-Module dar. Sie werden von ihm interpretiert und ausgeführt. Nachdem die Basismodule importiert wurden, meldet sich die interaktive Skriptanbindung mittels eines Informationstextes. Dem HADES-Benutzer steht von nun an der volle Funktionsumfang (siehe Abschnitt 8.1 auf Seite 57) der `jpshell` zur Verfügung.

Um die Struktur der Skriptanbindung besser zu verdeutlichen, dient das Sequenzdiagramm ihrer Initialisierung (s. Abb. 7.3), deren einzelnen Schritte anschließend erläutert werden. Auf die näheren Details der Implementierung wird im Kapitel 8 eingegangen.

Durch einen entsprechenden Menü-Eintrag im HADES-Editor kann ein Exemplar der `jpshell` erzeugt werden, eine Referenz des HADES-Editors wird ihr hierbei übergeben. Nachdem die `jpshell` ihre Oberfläche erzeugt hat, wird im eigentlichen ersten Schritt der

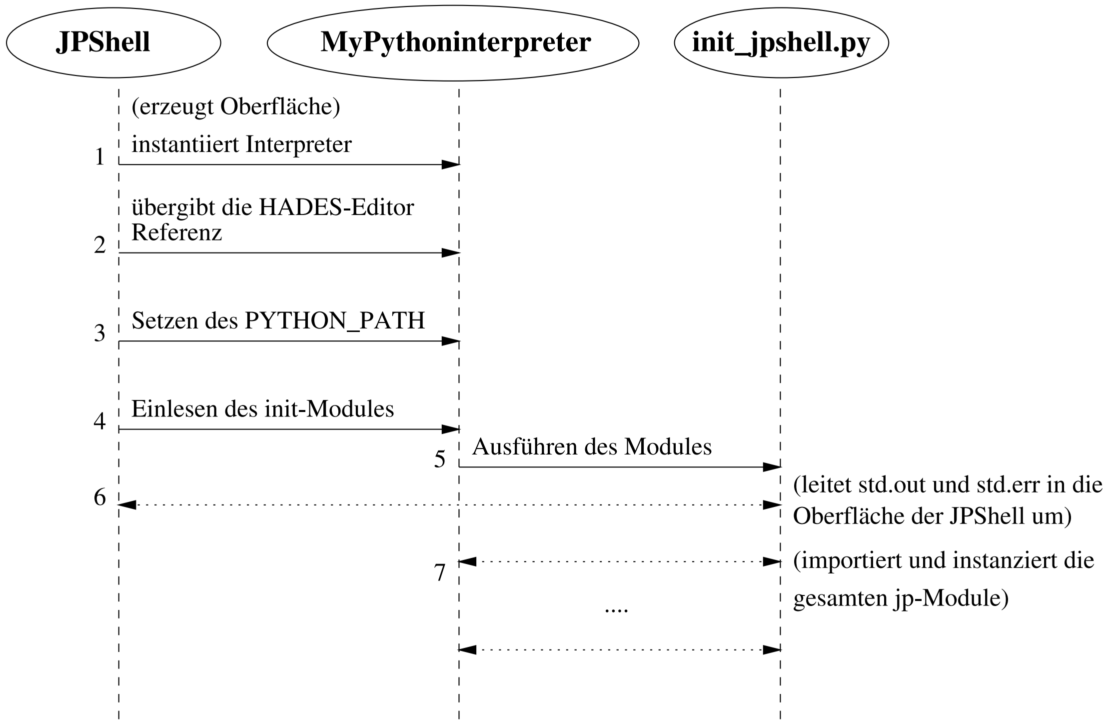

Abbildung 7.3: Die Initialisierung der Skriptanbindung wird durch die Erzeugung eines JPShell-Objektes durch den HADES-Editor angestoßen. Dieses instanziert einen Python-Interpreter, übergibt eine Referenz auf den HADES-Editor und setzt den Python-Pfad. Hiernach wird der Interpreter aufgefordert das Init-Modul einzulesen. Dieses leitet die Ausgaben des Interpreters in das GUI des JPShell-Objektes um und importiert alle Basis-Module.

Python-Interpreter initialisiert und die Umleitung der Standard Ein- und Ausgabe vorbereitet. Im zweiten Schritt wird die Referenz des HADES-Editor als globales Objekt im Interpreter hinterlegt. Dieses, im Folgenden `editor` genannte Python-Objekt, stellt die zentrale Bindung der Skriptanbindung an das laufende HADES-System dar. Nachdem im dritten Schritt der benötigte Arbeitspfad für Python erzeugt wurde, wird im vierten Schritt ein Python `init`-Modul ausgeführt. In diesem `init`-Modul werden die Standard Ein- und Ausgabe in die Oberfläche der `jpshell` umgelenkt und die Basismodule importiert. Hiernach meldet sich die `jpshell` mittels eines Begrüßungstextes und steht mit ihrer gesamten Funktionalität zur Verfügung.

7.2.2 Konzeption der Komponentenerweiterung von HADES

In diesem Abschnitt wird das Konzept für die Erweiterung von HADES durch neue Simulationsobjekte dargestellt. Deren Realisierung findet in Form von Python-Klassen statt, die dann mittels der `jpshell` einem HADES-Design hinzugefügt werden können. Es besteht ebenfalls die Möglichkeit, komplette Designs durch Python-Module zu erzeugen. Hiermit wird dem Anwender eine schnelle und mächtige Möglichkeit geschaffen, eigene Simulatio-

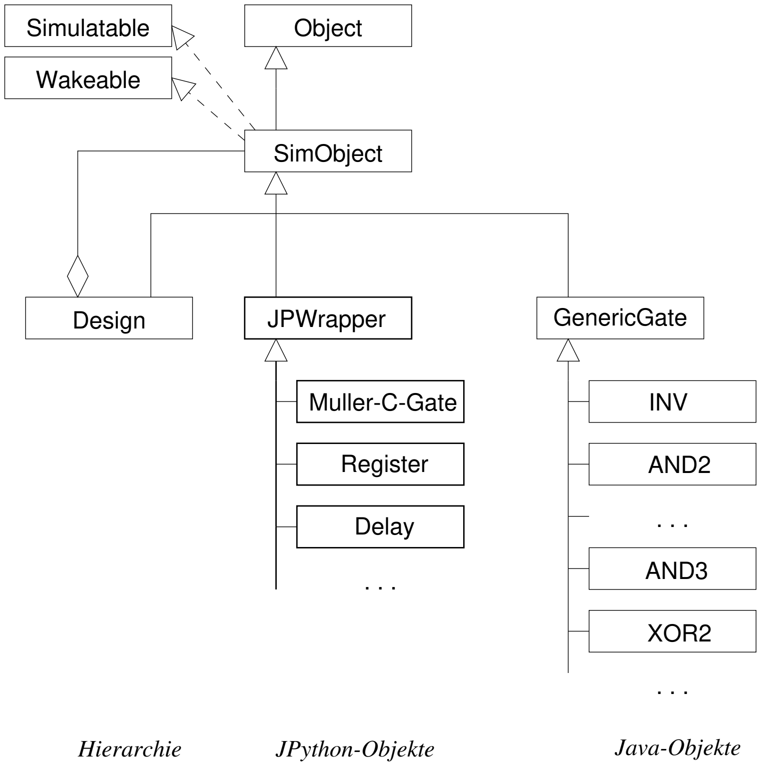

Abbildung 7.4: Dieses HADES-Klassendiagramm ordnet die JPythonWrapper-Klasse der Skriptanbindung in die bestehende Struktur ein. Hierbei stellt die JPythonWrapper-Klasse den Python-Komponenten dieselbe Funktionalität zur Verfügung, wie die GenericGate-Klasse ihren Java-Gattern.

nen interaktiv zu entwerfen. Das folgende Konzept, wird im Kapitel 10 auf Seite 75 durch eine Beispielimplementierung von asynchronen Komponenten konkretisiert.

Die bestehende HADES-Klassenhierarchie (s. Abb. 3.2 auf Seite 16) wurde hierfür um eine Java-Klasse erweitert. Diese `JPythonWrapper`-Klasse ist, analog zur `GenericGate`-Klasse, als Unterklasse von `SimObject` angelegt (s. Abb. 7.4).

Sie übernimmt zwei zentrale Aufgaben. Zum einen stellt sie die Superklasse der Python-Simulationsobjekte dar. Zum anderen enthält sie zur Laufzeit das entsprechende Python-Objekt, kapselt es, und tritt gegenüber dem Design vertretend auf. Im Bezug auf die Beispielimplementierung bedeutet dies, daß das Muller-C-Gate als Python-Klasse von der `JPythonWrapper`-Klasse erbt. Zur Laufzeit befindet sich ein Muller-C-Gate-Objekt innerhalb des `JPythonWrapper`-Objektes, dieses ist im Design enthalten und reicht die entsprechenden Aufrufe an das `Muller`-Objekt durch (s. Kapitel 10).

Diese Kapselung resultierte aus der Divergenz der Objektklassen von JPython und Java. Innerhalb von HADES wird wiederholt die Klasse eines Objektes durch die `getClass()`-Methode von Java ermittelt. Angewendet auf ein JPython-Objekt ist das Resultat die Klasse eines generierten Proxy-Objektes, aber nicht die eigentliche Python-Klasse. Dieser Mechanismus wird unter anderem beim Schreiben von Designs in eine HADES-Datei verwendet. Beim späteren Einlesen dieser Datei wird basierend auf der gespeicherten Klasse ein neues Objekt mittels des `newInstance()`-Konzeptes erzeugt. Hierbei wird nach der entsprechenden Java-Klasse im Klassenpfad gesucht und deren Konstruktor aufgerufen. Für gespeicherte JPython-Objekte existiert keine Java-Klasse, sondern jeweils nur das entsprechende Python-Modul.

Dieses sind die Hauptgründe für die Einführung der Java-Klasse `JPythonWrapper`, gleichzeitig ist durch diese Schnittstelle eine Änderung von HADES nicht nötig. Für ein Design ist ein `JPythonWrapper`-Objekt ein normales Simulationsobjekt, das Python-Objekt bleibt für die Anwendung transparent. Die Entwicklung von Simulationsobjekten durch JPython ist im Vergleich zur Java-Variante kürzer und schneller, da sie in der `jpshell` interaktiv und ohne neu zu übersetzen getestet werden können.

Kapitel 8

Realisierung der interaktiven Skriptsteuerung

8.1 Beschreibung des Funktionsumfanges

Eine Einführung in das Konzept und die Architektur der jpshell befindet sich im Abschnitt 7.2 auf Seite 52. Ihr Funktionsumfang wird durch die Basismodule der interaktiven Skriptsteuerung gebildet. Ein ausführliches Beispiel verdeutlicht deren Verwendung innerhalb der jpshell.

8.1.1 Vorstellung der Basismodule

Ziel dieses Abschnittes ist die Vorstellung der einzelnen Basismodule der HADES-Skriptanbindung. Ihre Funktionalität wird dargestellt, durch ausgewählte Implementierung und Beispiele soll anschließend das Verständnis vertieft werden.

Das Modul jp_design_os.py

Dieses Basismodul kapselt die Funktionalität, um interaktiv Design-Objekte innerhalb des HADES-Editors zu manipulieren. Zusätzlich werden Python-Module Funktionen für die Erzeugung von Designs zur Verfügung gestellt. Das Modul besteht aus drei Gruppen: die Erste enthält Hilfsmethoden, die den Zugriff auf HADES-Objekte regeln. Die zweite Gruppe stellt die Kommandos innerhalb der `jpshell` dar, und die letzte ist für die Erzeugung von Designs bestimmt.

- setEditor(editor);
  ersetzt das globale `editor`-Objekt durch das übergebende HADES-Editor-Objekt. Ausgehend vom `editor`-Objekt kann die Skriptanbindung auf alle Java-Methoden des HADES-Frameworks zugreifen.

- getEditor();
  gibt das aktuelle `editor`-Objekt zurück.

- updateDesign();
  ersetzt das globale `design`-Objekt (das aktuelle Simulationsexperiment), durch das sich im HADES-Editor befindliche Design-Objekt.

- getDesign();
  gibt das globale `design`-Objekt zurück. Sollte innerhalb des Interpreters kein `design`-Objekt existieren, so wird automatisch die `updateDesign`-Methode aufgerufen und dann dieses `design`-Objekt zurückgegeben.

- setDesign(design);
  ersetzt das globale `design`-Objekt durch das übergebene Design-Objekt.

- openDesign(resource); öffnet eine HADES-Design-Datei (`.hds`) im aktuellen HADES-Editor und aktualisiert das globale `design`-Objekt der HADES-Skriptanbindung. Die Datei sollte mit voll qualifizierten Namen angegeben werden.

Die zweite Gruppe des `jp_design_os`-Moduls beinhaltet die eigentlichen `jpshell`-Kommandos, die auf dem `design`-Objekt der `jpshell` operieren. Sie sind teilweise an bekannte Shell-Kommandos angelehnt. Hierarische Designs sind in HADES ähnlich eines UNIX-Dateisystems organisiert, ausgegehend von einem Wurzeldesign sind die Subdesigns und die einzelnen Simulationsobjekte zu ereichen. Sie werden im Folgenden beschrieben und durch die anschließenden Beispiele in ihre Verwendung illustriert.

- infoDesign();
  gibt Informationen über das aktuelle `design`-Objekt aus. Sie bestehen aus dem Namen des Designs und einer Auflistung seiner Komponenten, mit Namen und Typ.

- pwd();
  gibt den aktuellen Designpfad aus und zurück, z.B. `/prima/prima-control` bezeichnet das Steuerwerk des PRIMA-Prozessors.

- ls(pattern=“”);
  gibt alle Simulationskomponenten aus, die sich innerhalb des aktuellen Designpfades befinden. Optional kann ein regulärer Ausdruck angegeben werden, um das Ergebnis einzuschränken.

- cd(design_name);
  ändert den aktuellen Designpfad auf das übergebende Subdesign.

- getByName(name);
  gibt die Simulationskomponente zurück, die durch den übergebenden Namen spezifiziert wurde.

- get(pattern=“”);
  gibt alle Simulationskomponenten innerhalb des aktuellen Designpfades in Form eines Java-Vektors zurück. Optional kann ein regulärer Ausdruck angegeben werden.

Für die Erstellung von Design-Objekten durch Python-Module sind die folgenden Funktionen gedacht. Sie stellen eine dritte Gruppe im `jp_design_os`-Modul dar.

- addComponent(component, c_name=None);
  fügt ein `JPythonWrapper`-Objekt (s. Abschnitt 10.2.1 auf Seite 78) dem Design hinzu, dieses kapselt die übergebende Komponente. Optional kann ein Objektname angegeben werden.

- addComponentWithSymbol(component, pos, c_name=None, mirror=None);
  erweitert die `addComponent`-Funktion um das Erzeugen eines graphischen Symbols für die Komponente. Dieses kann mittels eines zusätzlichen Parameters gespiegelt werden und wird anschließend dem HADES-Editor übergeben.

- connect(sender, sender_port_name, receiver, receiver_port_name, signalName=None);
  verbindet den angegebenen Senderport mit dem Empfängerport durch ein Signal, dessen Name optional spezifiziert werden kann.

- connectWithSignal(sender, ... , signalName=None, wire_points=None);
  verbindet die Komponenten wie die `connect`-Funktion, zusätzlich wird das Signal graphisch im HADES-Editor angezeigt. Dessen Erscheinungsform kann durch die Angabe von zusätzlichen Wegpunkten manipuliert werden.

- connectWithSignalPos(pos_sender, ... , signalName=None, wire_points=None);
  stellt eine Alternative zur `connectWithSignal`-Funktion dar, sowohl der Sender als auch der Empfänger werden über ihre Positionen im HADES-Editor identifiziert. Dieses ist bei Designs nötig, die die automatische Benennung von Simulationsobjekten durch HADES verwenden.

- connectToSignal(obj, obj_port, target_signal, position);
  verbindet eine Komponente mit einem bestehenden Signal über die angegebenen Wegpunkte.

- findSimObject(position);
  sucht innerhalb des HADES-Editors nach einem Simulationsobjekt an der spezifizierten Position und gibt dieses zurück.

- findSignal(position);
  liefert analog zur `findSimObject()`-Funktion das gefundene Signal.

Das Modul jp_os.py

Dieses Basismodul ermöglicht es der `jpshell` auf das Dateisystem des Rechners zuzugreifen. Der größte Teil der Methoden entsprechen den üblichen Shell-Kommandos.

- mkdir(dir_name);
  erzeugt im aktuellen Verzeichnis ein neues Unterverzeichnis.

- rm(file);
  löscht die angegebene Datei oder Verzeichnis.

- rename(file, new_file);
  benennt die angegebene Datei oder das Verzeichnis in den neuen Namen um.

- pwd();
  gibt den aktuellen Pfad aus und zurück.

- getPath();
  ist eine Hilfsmethode, die den aktuellen Pfad zurückgibt.

- setPath(new_path);
  ist eine Hilfsmethode, um den aktuellen Pfad direkt zu setzen.

- ls();
  listet alle Dateien und Verzeichnisse im aktuellen Pfad auf.

- cd(new_path);
  wechselt in das übergebende Verzeichnis und aktualisiert den Pfad.

- stat(file);
  gibt die Statusinformationen der übergebenen Datei aus. Das Ausgabeformat gibt an, ob man die Rechte zum Lesen und Schreiben besitzt, desweiteren noch die Dateigröße und das Datum der letzten Änderung.

Das Modul jp_tools.py

Hilfsmethoden, deren Funktionalität allgemein benutzt werden soll, aber nicht zu einem Themenkomplex gehört, sind in diesem Modul zusammengefaßt.

- infoObject(object, pattern="");
  gibt unter Benutzung der Java-Reflection-API alle Methoden des übergebenen Objektes aus. Optional kann ein regulärer Ausdruck angegeben werden.

- getHelp(module, pattern="");
  gibt für jede Methode des übergebenen Moduls einen Hilfetext aus. Ein regulärer Ausdruck kann als optionaler Parameter mit übergeben werden.

- welcome();
  wird beim Starten der `jpshell` ausgeführt und gibt einen Informationstext und Versionsinformationen aus.

Das Modul jp_meta_tools.py

In diesem Modul befinden sich Funktionen, die auf anderen Modulen arbeiten.

- help(pattern="");
  gibt alle Methoden aus allen Basismodulen aus, die zu dem übergebenen regulären Ausdruck passen.

- doReload();
  verwendet die `reload()`-Funktion von Python, um alle Basismodule zur Laufzeit aktualisieren zu können.

Das Modul jp_sim_control.py

Dieses Modul enthält einige Hilfsmethoden, um mit dem HADES-Simulator zwecks Simulationssteuerung interagieren zu können.

- getSimulator();
  gibt das globale `simulator`-Objekt zurück.

- setSimulator(sim);
  mittels dieser Hilfsmethode wird der übergebende HADES-SimKernel als globales `simulator`-Objekt gespeichert.
- updateSimulator();
  ermittelt durch das `editor`-Objekt den HADES-SimKernel und ersetzt das `simulator`-Objekt durch ihn.
- pauseSimulator();
  versetzt den HADES-Simulator in den Pause-Zustand.
- resetSimulator();
  die Simulation wird in ihren Anfangszustand versetzt.
- runSimulator(time=”);
  läßt die Simulation unendlich lange weiter laufen, es sei denn eine Laufzeit wurde explizit angegeben.

8.2 Beispiel für die Verwendung der jpshell

Das folgende Beispiel verdeutlicht die Verwendung und die Möglichkeiten der jpshell. Mit Hilfe von einfachen Kommandos wird ein bestehendes Design manipuliert, durch den Design-Baum navigiert und auf Simulationsobjekte zugegriffen.

Ausgangssituation

Durch den Datei-Öffnen Dialog des HADES-Editors wurde das `prima`-Design (s. Abb. 1.1 auf Seite 4) geladen. Eine jpshell kann über den Menüpunkt 'Special/Open JPython shell' erzeugt werden. Nach erfolgreicher Initialisierung der jpshell meldet diese sich mittels eines Infotextes (s. Abb. 8.1).

Das vorher geöffnete `prima`-Design wurde bereits der jpshell übergeben und der Benutzer kann sofort die Funktionalität der Shell verwenden. Informationen über die einzelnen Kommandos sind durch die `help`-Funktion zu bekommen.

Einfache Shell-Kommandos

Im ersten Schritt wird durch die Eingabe des Kommandos `help(‘‘ls’’)`, der Hilfstext des `ls`-Befehls ausgegeben (s. Abb. 8.2).

Hierbei ist zu beachten, daß zwei Hilfetexte ausgegeben wurden. Das Argument des help-Kommandos ist ein regulärer Ausdruck und es existieren zwei Befehle mit dem Namen `ls`. Diese befinden sich in verschiedenen Modulen, das zweite `ls`-Kommando stammt aus dem `jp_os`-Modul und ist mittels Punkt-Notation voll qualifiziert anzugeben. Für dieses Beispiel ist das erste `ls`-Kommando von Interesse. Dies gibt alle auf der oberstes Ebene liegenden Simulationsobjekte des aktuellen Designs aus.

Die resultierende Ausgabe (s. Abb. 8.3) gibt für jedes Simulationsobjekt zuerst dessen Namen, gefolgt vom Typ und zuletzt den Design-Pfad des Objektes aus. Besondere Beachtung ist der `prima-control`-Komponente zu schenken. Hierbei handelt es sich um ein eigenes Subdesign. Im nächsten Schritt wird durch das `cd`-Kommando (s. Abb. 8.4) in dieses Design gewechselt.

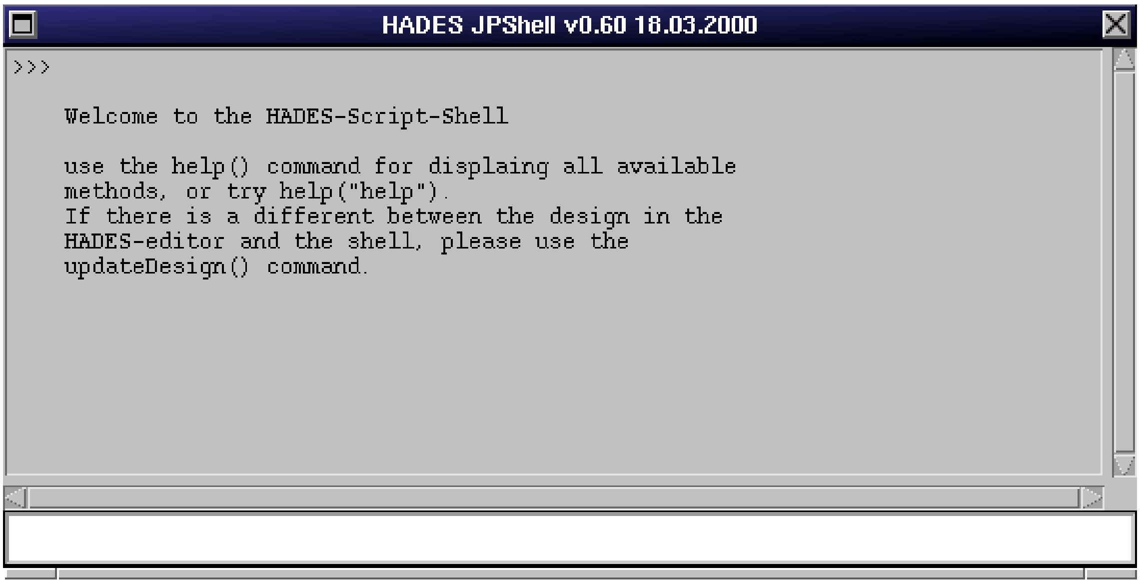

Abbildung 8.1: Die Startausgabe der jpshell informiert über die verwendete Version und macht auf das help-Kommando aufmerksam.

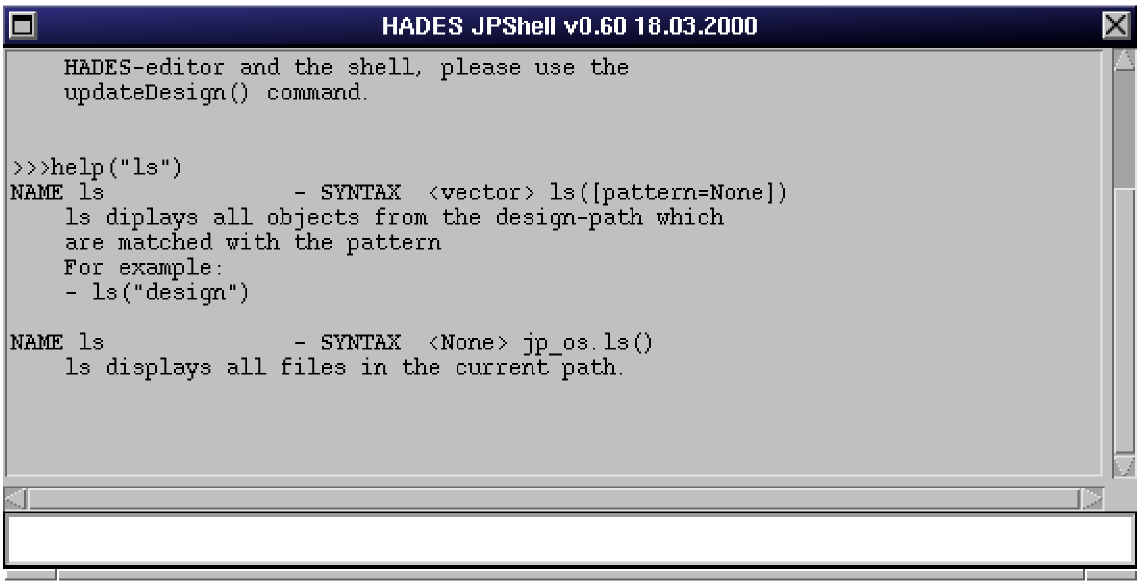

Abbildung 8.2: Die Onlinehilfe des ls-Kommandos wird als Resultat eines help("ls")-Kommandos in der jpshell angezeigt. Die Syntax des Befehls, eine kurze Erklärung und ein Beispiel verdeutlichen die Verwendung der Befehle. Das Argument des help-Kommandos ist ein regulärer Ausdruck deshalb werden die beiden ls-Befehle aus unterschiedlichen Basis-Modulen angezeigt.

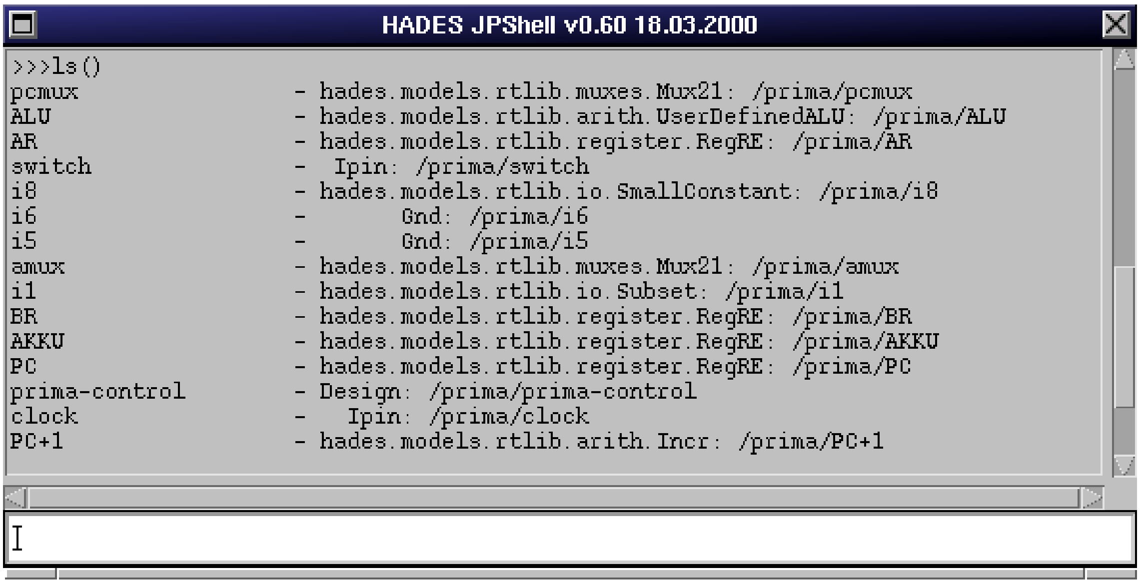

Abbildung 8.3: Mit Hilfe des ls-Kommandos werden alle Simulationskomponenten des aktuellen Designs ausgegeben. Hierbei wird der Name der Komponente, dann ihre Klasse und schließlich die Design-Hierarchieebene angezeigt.

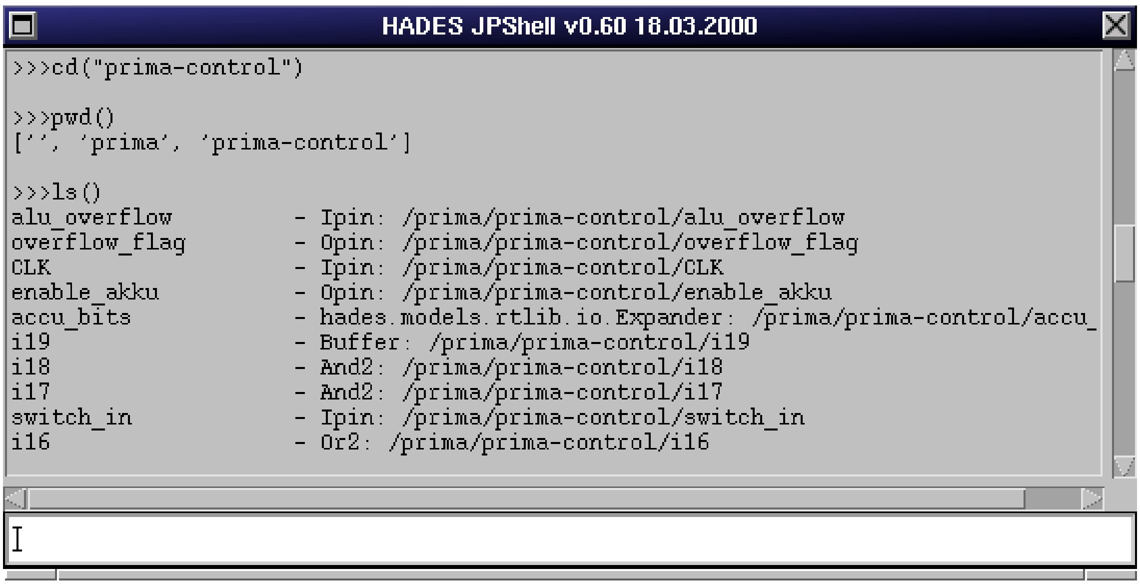

Abbildung 8.4: Mittels eines cd-Kommandos kann in ein Subdesign gewechselt werden. Der pwd-Befehl gibt den aktuellen Designpfad aus. Die folgenden Kommandos beziehen sich von jetzt an auf das Subdesign. Dieses wird nicht im HADES-Editor angezeigt, kann aber durch die jpshell direkt manipuliert werden. Eine wichtiges Problem des HADES-GUIs ist damit gelöst.

Die Kommandos in der jpshell beziehen sich von nun an auf das `prima-contol`-Subdesign. Das `pwd`-Kommando gibt den aktuellen Design-Pfad aus. Ein erneutes `ls`-Kommando listet nun alle Simulationsobjekte innerhalb des Subdesigns auf. Dieses ist eine wesentliche Verbesserung gegenüber der graphischen Oberfläche. Es können direkt Design-Ebenen manipuliert werden, ohne diese erst anzeigen zu müssen. Wie von einem Dateisystem gewohnt, kann durch ein `cd("..")`-Befehl wieder eine Ebene nach oben gewechselt werden.

Zugriff auf Simulationsobjekte

Nachdem gezeigt wurde, wie man Komponenten eines Design auflistet und wie innerhalb des Design-Baumes navigiert werden kann, wird jetzt direkt auf Simulationsobjekte zugegriffen. Die jpshell kann auf jedes Objekt innerhalb des Python-Interpreters zugreifen, deshalb ist die Erzeugung eines Python-Objekte immer der erste Schritt,. Hierfür stehen aus dem `jp_design_os`-Modul zwei `get`-Funktionen bereit (s. Seite 58). Um gezielt eine Referenz auf genau ein Simulationobjekt in einem Python-Objekt abzulegen, wird die `getByName`-Funktion verwendet (s. Abb. 8.5).

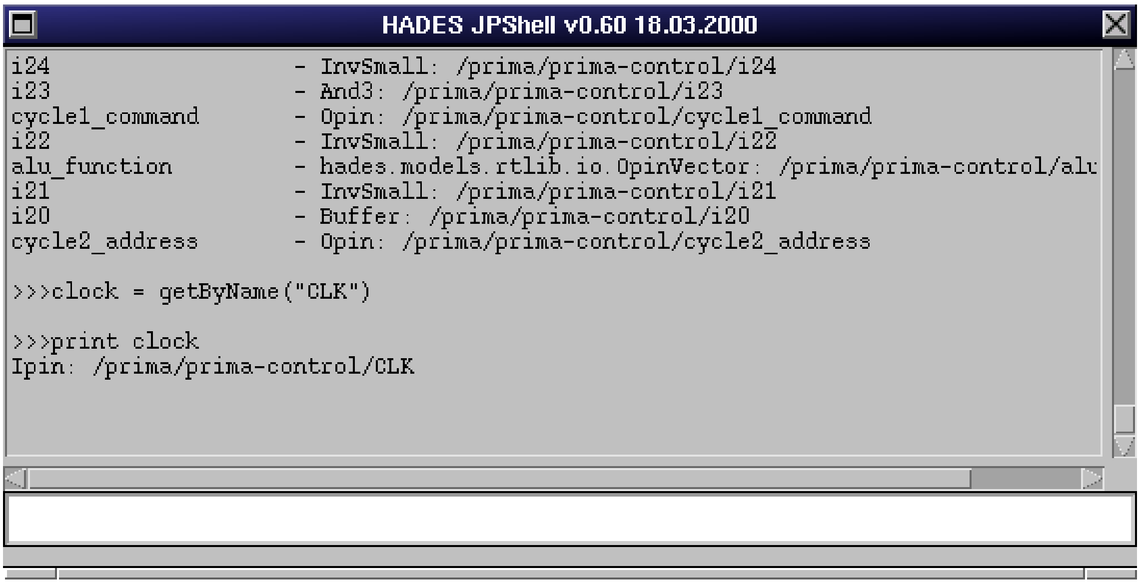

Abbildung 8.5: Damit Simulationsobjekte direkt innerhalb der jpshell manipuliert werden können, muß zuerst ein lokales Objekt angelegt werden. Hierfür stehen verschiedene get-Funktionen bereit, die getByName-Funktion sucht im Design nach dem spezifizierten Objekt und gibt es zurück. Hiernach kann darauf normal zugegriffen werden, z.B. können sie mittels des Python print-Befehls ausgegeben werden.

Die Python-Anweisung `clock = getByName("CLK")` erzeugt ein neues Python-Objekt `clock` im Interpreter. Dessen Wert und Typ wird vom Ergebnis des Ausdrucks der rechten Seite der Zuweisung festgelegt. In diesem Beispiel liefert die `getByName`-Funktion einer Referenz auf das `CLK`-Simulationsobjekt zurück. Auf das neue `clock`-Objekt kann sowohl lesend und schreibend zugegriffen werden. Die nachfolgende `print`-Anweisung gibt den

String aus, der in der `toString`-Methode der entsprechenden Java-Klasse spezifiziert wurde. Mit Hilfe der *Introspection*-Eigenschaft der Reflection-API von Java, kann nun durch das `infoObject`-Kommando, z.B. alle `get`-Methoden des `clock`-Objektes ausgegeben werden (s. Abb. 8.6)

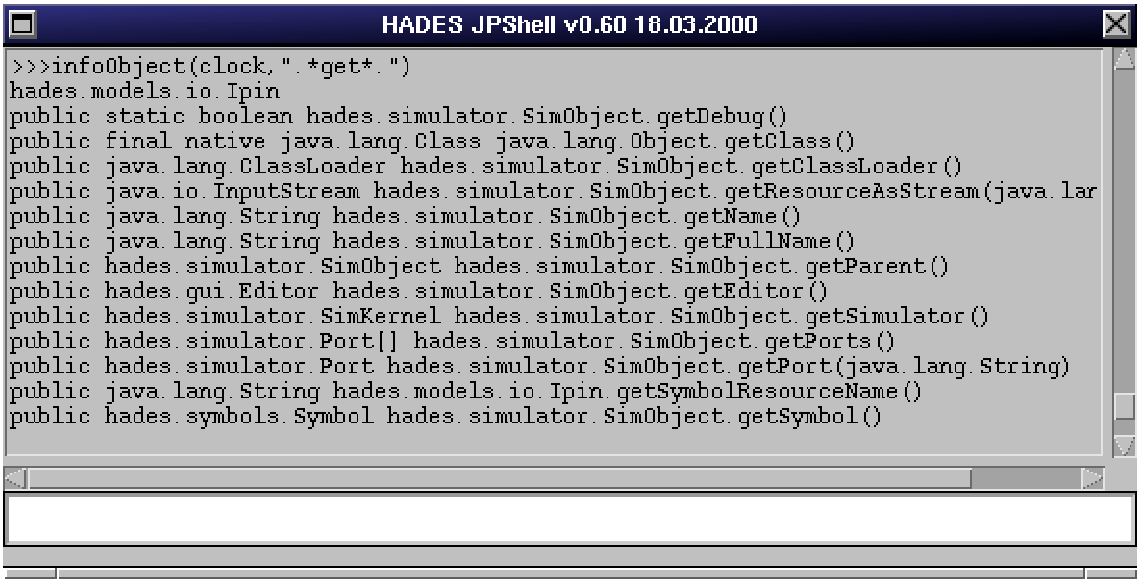

Abbildung 8.6: Durch die Reflection-API können gewünschte Methoden eines Objektes innerhalb der jpshell aufgelistet werden. Dieses wird durch das infoObjekt-Kommando realisiert, zusätzlich kann ein regulärer Ausdruck angegeben werden. In diesem Beispiel werden alle get-Methoden des clock-Objektes ausgegeben.

Alle Methoden, die mittels des `infoObject`-Kommandos aufgelistet werden, sind als `public`-deklariert und können auf das Objekt angewendet werden. Hierdurch konnte die Verzögerung der Simulationskomponente ermittelt werden (s. Abb. 8.7). Es sei nochmal betont, daß sich dieses Objekt innerhalb eines Subdesigns befindet und nicht graphisch angezeigt wird. Objekte können auch direkt verändert werden, so ist ebenfalls in der Abbildung 8.7 die Manipulation der Verzögerung dargestellt.

Komplexere Shell-Programmierung

Einzelne Simulationsobjekte können meist auch durch die graphische Benutzeroberfläche des HADES-Editors manipuliert werden. Das abschließende Beispiel (s. Abb. 8.8) zeigt sehr schön die Vorteile der Kommandozeile.

Die jpshell unterliegt der normalen Python-Syntax und somit kann in ihr auch programmiert werden. Eine einfache `for`-Schleife ändert die Verzögerungszeit aller UND-Gatter im Subdesign. Komplette Python-Module können durch einfaches Importieren ausgeführt werden, auf diesem Wege können HADES-Benutzer das System nach ihren Bedürfnissen anpassen. Die Funktionalität der HADES-STL und der Komponentenerweiterung durch die Skriptanbindung wird durch entsprechende Module realisiert, die aus der jpshell heraus gestartet werden.

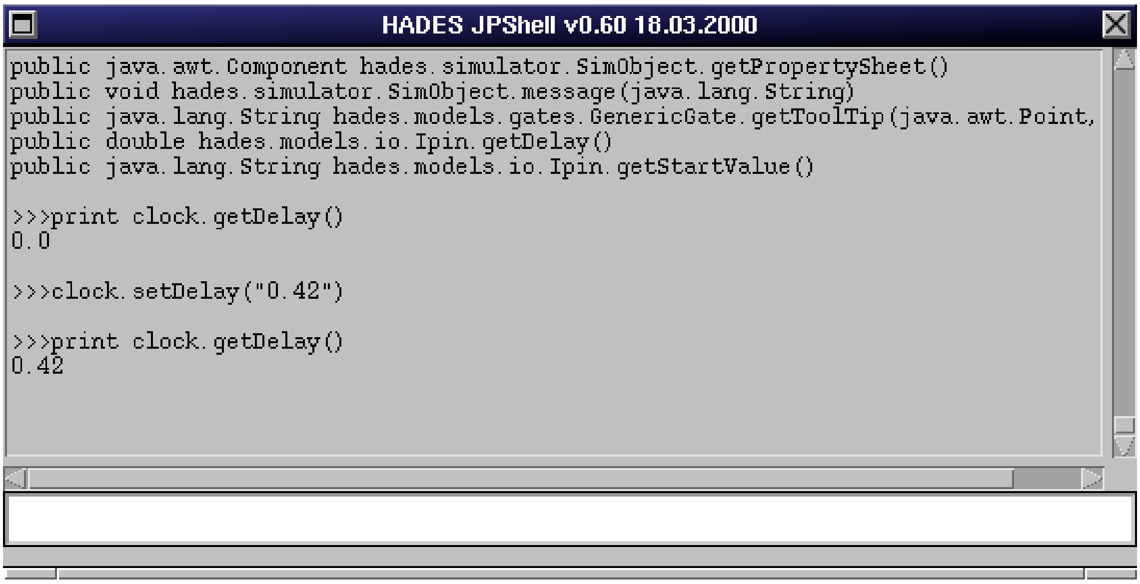

Abbildung 8.7: Auf lokale JPython-Objekte der jpshell können alle public-Methoden direkt angewendet werden. So ist auf einfache Weise die Verarbeitungszeit einer Komponente zu ändern. Das entsprechende Simulationsobjekt wird direkt verändert, das lokale Objekt ist eine Objekt-Referenz.

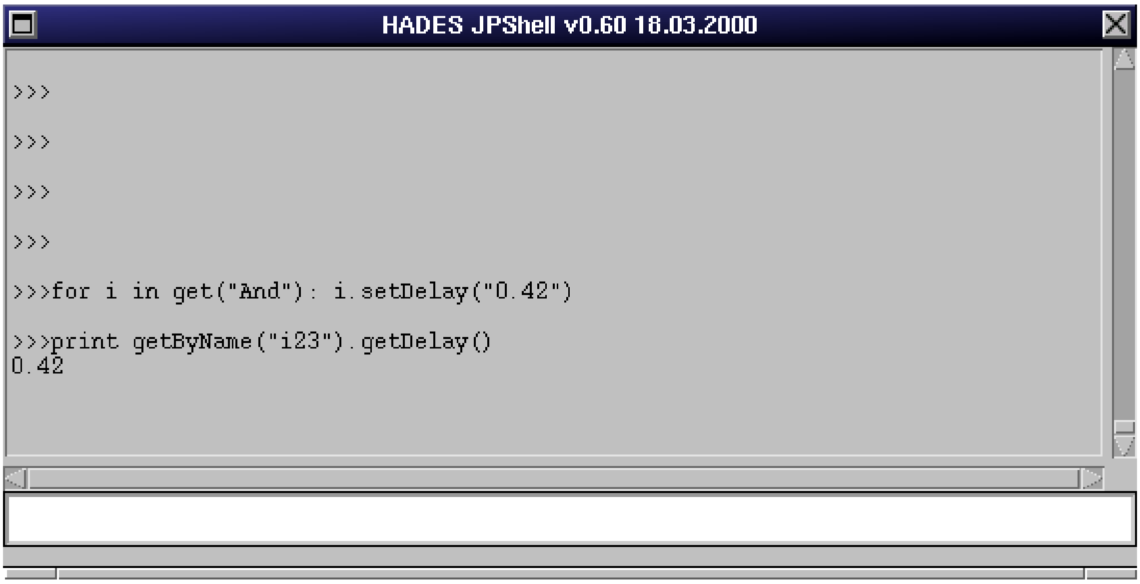

Abbildung 8.8: Die jpshell unterliegt der Python-Syntax. Ihre Kontrollstrukturen, wie z.B. Schleifen, können zur Programmierung innerhalb der jpshell benutzt werden. Diese macht die jpshell zu einem mächtigen und sehr effizienten Werkzeug für HADES. Arbeitsintensive Veränderungen, wie z.B. die Verarbeitungszeit aller UND-Gatter eines kompletten Designs zu ändern, können durch eine einfache Schleife realisiert werden. Dies verdeutlicht die Vorteile einer Kommandozeile gegenüber einer graphischen Oberfläche.

Kapitel 9

Realisierung der HADES Simulations- und Test-Sprache

9.1 Einführung und Aufbau der HADES-STL

9.1.1 Einführung STL

Die Simulations- und Test-Sprache von HADES erlaubt die Beschreibung von Simulationsstimuli für Designs, die später vom Simulator in Eingabewerte überstetzt werden. Durch die Festlegung von Abtastpunkten können auch zu erwartende Ergebnisse überprüft werden. Durch spezielle STL-Module können Designs mit definierten Stimuli getrieben und getestet werden. Hierfür steht die im Abschnitt 9.2 beschriebende Funktionalität zur Verfügung, die sich an die STL des CADENCE-Simulators anlehnt [Cad97]. Um sie dem Vorbild auch sprachlich anzupassen, wurde sie HADES-STL genannt, es handelt sich jedoch im strengeren Sinne nicht um eine eigene Sprache. Vielmehr umfaßt sie die Erweiterung der HADES-Skriptanbindung um die Funktionalität einer CADENCE-STL.

9.1.2 Aufbau eines STL-Moduls

Im Folgenden werden die wichtigsten Bestandteile eines STL-Moduls beschrieben, ein vollständiges STL-Modul befindet sich im ausführlichen Beispiel am Ende dieses Kapitels (s. Seite 72). Ein STL-Modul ist ein normales Python-Modul (Datei mit der Endung `.py`) und sollte sich in einem Verzeichnis befinden, das über den Pythonpfad (`sys.path`) erreichbar ist. Da es sich um ein JPython-Modul handelt steht natürlich die volle Python- und Java- Funktionalität zur Verfügung. STL-Skripte unterliegen der Python-Syntax und deren Kontrollstrukturen.

Ein solches Modul gliedert sich in fünf Abschnitte:

1. Initialisierung
   Der erste Schritt der Initialisierung besteht im Öffnen des entsprechenden HADES-Designs. Diese werden in HADES-Dateien (Endung `.hds`) gespeichert und sind mit vollem Pfadnamen anzugeben. Hiernach ist die Laufzeitumgebung der HADES-Skriptanbindung zu aktualisieren und anschließend wird das `VectorGen`-Objekt erzeugt.

```
# Initialisierung
jp_design_os.openDesign(<HADES-Datei>)
jp_design_os.updateDesign()
jp_sim_control.updateSimulator()
<VectorGen-Objekt> = VectorGen()
```

2. Definition der Ein- und Ausgänge
   Mit Hilfe der in Abschnitt 9.2.1 auf Seite 69 vorgestellten `defPin`-Prozeduren werden lokale Objekte erzeugt. Sie stellen Referenzen auf die Simulationsobjekte innerhalb des Simulators dar.

3. Definition der Abtastpunkte
   Für das Festlegen von Erwartungswerten innerhalb der Stimulidefinition ist es notwendig, daß die entsprechenden Referenzen auf die Abtastpunkte bestehen. Sollen keine Erwartungswerte während der Ausführung der Stimuli getesten werden, ist dieser Abschnitt nicht notwendig. Bevor auf die Abtastpunkte zugegriffen werden kann, müssen diese dem Design hinzugefügt werden. Hierfür stehen mehrere Modi zur Verfügung (s. Seite 69).

   Nachdem die Abtastpunkte angelegt sind, können die lokalen Objekte erzeugt werden. Für diese Funktionalität stehen mehrere `defProbe`-Prozeduren zur Verfügung. Die Werte der Abtastpunkte können während der Simulation durch den HADES-`Waveform Viewer` verfolgt werden.

```
# Definition der Abtastpunkte
addProbes(<mode>)

# Es wird für jeden Abtastpunkt ein Referenz-Objekt angelegt.
<Liste aller Abtastpunkt> = defAllProbes()

# Es werden nur die benötigten Abtastpunkte referenziert.
<Abtastpunkt> = defProbesBevore(<Simulationsobjekt>)
<Abtastpunkt> = defProbesAfter(<Simulationsobjekt>)
```

4. Definiton der Stimuli
   Während die ersten drei Abschnitte eher den statischen Teil mit der nötigen Vorbereitung darstellen, folgt nun der dynamische Teil, der sich aus den Methoden der `hades.jpshell.VectorGen`-Klasse zusammensetzt (s. Seite 70). Im ersten Schritt wird das zu verwendende Zeitschema für die Stimuli festglegt. Hierzu wird eine Startzeit und eine Intervallänge angegeben. Zu Beginn des Intervalls werden die Stimuli an den Eingängen angelegt und am Ende findet die Prüfung der Erwartungswerte statt.

```
<VectorGen-Objekt>.defTiming(<Startoffset>, <Intervallänge>)
```

Hiernach wird das Eingabeformat festgelegt. In der hier festgelegten Reihenfolge werden später die einzelnen Stimuli ausgeführt. Befinden sich am Ende der Liste Abtastpunkte, so wird der entsprechende Eintrag als Test interpretiert.

```
<VectorGen-Objekt>.defFormat([
  <Objekt-Liste>[<Abtastpunkt-Liste>]
])
```

Im letzten Schritt werden die Stimuli definiert. Sie werden einfach in einer Liste zusammengefaßt. Es stehen die logischen Stimuli `0`, `1`, `X` (don't care) und `U` (undefined) zur Verfügung. Zu beachten ist hierbei, daß die Länge der Stimuliliste einem Vielfachen der Anzahl der Objekte im Format entspricht. Die Erwartungswerte der Abtastpunkte können explizit oder in Form eines boolschen Ausdrucks in präfix-Notation angegeben werden.

```
<VectorGen-Objekt>.defStimuli([
  <Stimuli-Liste>[<Erwartungswerte-Liste>]
])
```

5. Starten der Stimuliverarbeitung
   Nach Abschluß der Vorbereitungen befinden sich alle Stimuli und Erwartungswerte innerhalb des `VectorGen`-Objektes, das diese nun der Reihe nach abarbeitet.

9.2 Beschreibung des Funktionsumfanges

Die HADES-STL besteht aus dem `jp_stl`-Modul der Java-Klasse `VectorGen` und der `Stimuli`-Hilfsklasse.

9.2.1 Vorstellung des statischen Anteils der STL

Für die Erstellung von STL-Modulen ist ein einfacher Zugriff auf Designobjekte wichtig. Lokale Python-Objekte müssen erzeugt und die Signale des HADES-Designs mit Abtastpunkten versehen werden. Diese statische Funktionalität bietet das `jp_stl`-Basismodul.

Funktionserklärung des Moduls: jp_stl.py

- defAllPins(pattern);
  liefert eine Liste aller im Design vorhandenen Ein- oder Ausgangsgatter zurück, die dem übergebenen regulären Ausdruck entsprechen.

- defPinByName(name);
  gibt das Ein- oder Ausgangsgatter zurück, das durch den übergebenden Namen spezifiziert wurde. Wird kein entsprechendes Gatter gefunden, wird ein Python `none`-Wert zurückgegeben.

- addProbes(mode);
  erzeugt für Signale im Design die entsprechenden Abtastpunkte, dabei stehen folgende Modi zur Verfügung:
  - ALL: Für jedes Signal in allen Hierarchieebenen des aktuellen Designs wird ein Abtastpunkt erzeugt.
  - TOPLEVEL: Es werden nur die Signale der abgebildeten obersten Hierarchieebene mit Abtastpunkten versehen.
  - TOPLEVELIO: Nur für die Signale der obersten Ein- oder Ausgängen des Designs werden Abtastpunkten erzeugt.
- addProbeByName(signal_name);
  das Signal, das dem Parameter entspricht, wird mit einem Abtastpunkt versehen.
- defAllProbes();
  liefert eine Liste aller sich im Design befindenden Abtastpunkte zurück.
- defProbeByName(name);
  gibt das entsprechende Abtastpunkt-Objekt zurück.
- defProbesAfter(component);
  liefert eine Liste aller Abtastpunkte zurück, die zu den Signalen hinter dem entsprechenden Simulationsobjekt gehören. Sollte nur ein Abtastpunkt gefunden werden, so wird dieses Objekt direkt zurückgegeben.
- defProbesBevore(component);
  liefert eine Liste aller Abtastpunkte zurück, die zu den Signalen gehören, die sich vor dem spezifizierten Simulationsobjekt befinden. Wurde nur ein Abtastpunkt gefunden, so wird dieses Objekt direkt zurückgegeben.
- printAllProbes(time);
  es werden alle Abtastpunkte des Designs mit ihrem Wert zur Simulationszeit time ausgegeben.
- addClock(ipin, offset, period, dutycycle);
  ersetzt den übergebenen Design-Eingang durch einen Taktgeber mit den spezifizierten Attributen.

9.2.2 Vorstellung des dynamischen Anteils der STL

Simulationsexperimente müssen mit Stimuli getrieben, und der Wert von Abtastpunkten muß ermittelt werden. Diese Funktionalität stellt die folgende Java-Klasse den STL-Modulen zur Verfügung.

Funktionserklärung der VectorGen-Klasse

Diese JPython-Klasse kapselt den dynamischen Anteil der HADES Simulations- und Test-Sprache. Im Folgenden werden die öffentlichen Methoden der `hades.jpshell.VectorGen`-Klasse erläutert. Sie gliedern sich in zwei Bereiche; die Integration in das HADES-System und die Methoden für die Erstellung der STL-Module.

- Für die Integration in das HADES-System sind vorgesehen:
  - elaborate(self, arg);
    wird vom Simulator beim Hinzufügen zu einem Design einmal aufgerufen. Das Simulationsobjekt erhält eine Referenz auf den Simulator und erzeugt zwei Wakeup-Events. Diese werden in der Ereignisliste des Simulators eingetragen. Das erste Simulationsevent stößt die Ausführung der Stimuli an und das zweite beginnt den Testzyklus.
  - evaluate(self, arg);
    wird für die Simulationssteuerung benötigt, beinhaltet aber keine Simulationslogik.
  - wakeup(self, arg);
    wird bei der Ausführung eines Wakeup-Events durch den Simulator aufgerufen. Der Simulator übergibt den gewünschten Modus als Argument. Es gibt die beiden folgenden Modi:
    * APPLY-Modus: Das `VectorGen`- Objekt führt die nächsten Stimuli aus und trägt ein neues Wakeup-Event in den Eventstack ein.
    * TEST-Modus: Das VectorGen Objekt führt den im STL-Format angegebenen Test aus, und ein neues Test-Wakeup-Event wird erzeugt.
- Die folgenden Methoden sind für die Erstellung von STL-Modulen vorgesehen:
  - defTiming(self, start, end);
    definiert das zu verwendende Zeitschema.
  - getTiming(self);
    das aktuelle Zeitschema-Intervall wird zurückgegeben.
  - defFormat(self, format);
    definiert das STL-Format für die Ausführung der Stimuli und Tests in Form einer Objektliste.
  - getFormat(self);
    gibt das bestehende STL-Format als Liste zurück.
  - defStimuli(self, stimuli_list);
    definiert die auszuführenden Stimuli und Tests. Sie werden in Form einer Stimuli-Liste übergeben, deren Reihenfolge mit dem angegebenen Format harmonieren muß.
  - executeStimulus(self, stimulus, debug=0);
    das übergebende Stimulus-Objekt beinhaltet unter anderem das Simulationsobjekt, auf das der Stimulus zur definierten Simulationszeit angewendet wird.
  - start(self);
    die Ausführung der STL wird gestartet.
  - getResult(self);
    es wird eine Liste der Testergebnisse zurückgegeben.
  - testProbe(self, probe, state, time);
    gibt den Wert des Abtastpunkte, zur Simulationszeit `time` zurück.

9.3 Beispiel eines STL-Moduls

Dieses Beispiel verdeutlicht Arbeitsweise und Möglichkeiten der HADES-STL. Das STL-Modul wird durch ein `import`-Kommando aus der jpshell heraus gestartet und öffnet zunächst das HADES-Design `dlatch` (s. Abb. 9.1).

An die Initialisierung anschließend werden die lokalen JPython-Objekte für die Ein- und Ausgänge des Designs erzeugt. Durch die `addProbes`-Funktion Zeile 18 werden alle Signale mit einem Abtastpunkt versehen. Das Format der STL-Stimuli wird mittels der lokalen Objekte spezifiziert (Zeile 28-30). In diesem Beispiel hat das STL-Format die Länge 3 und der erste Stimulus wird auf den Eingang C angewendet. Der Eingang D erhält den zweiten Stimulus, dies geschieht gleichzeitig zur Simulationszeit 0,0. Beim dritten Eintrag im Format handelt es sich um den Abtastpunkt *probeQ*, dessen Wert wird zum Ende des Zeitintervalls mit dem des Stimulus verglichen.

stl_dlatch.py

```
 1  import jp_design_os
 2  import jp_sim_control
 3  from jp_stl import *
 4  from VectorGen import *
 5
 6  # 1. Initialisierung
 7  jp_design_os.openDesign("/hades/examples/simple/dlatch.hds")
 8  jp_design_os.updateDesign()
 9  jp_sim_control.updateSimulator()
10  v = VectorGen()
11
12  # 2. Definition der Ein- und Ausgaenge
13  pinC = defPinByName("C")
14  pinD = defPinByName("D")
15  pinQ = defPinByName("Q")
16
17  # 3. Definition der Abtastpunkte
18  addProbes(mode="ALL")
19  probeQ, pNQ, pI1Y, pI0Y, pD, pC = defAllProbes()
20
21  # Alternativ
22  # probeQ = defProbesBevor(Q)
23  # pC = defProbesAfter(C)
24
25  # 4. Definiton der Stimuli
26  v.defTiming(start=0.0, end=1E-8)
27
28  v.defFormat([
29    pinC, pinD, probeQ
30  ])
31
```

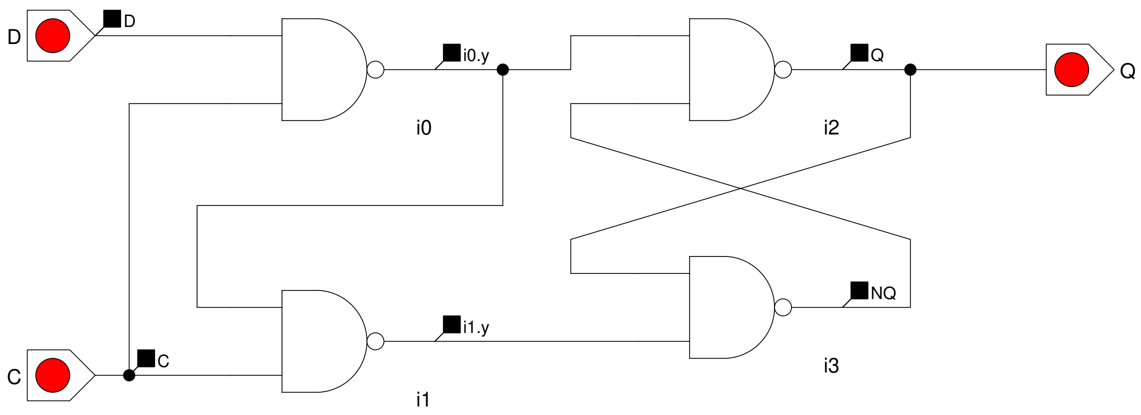

Abbildung 9.1: Das dlatch-Design simuliert ein pegelgesteuertes D-Flipflop.

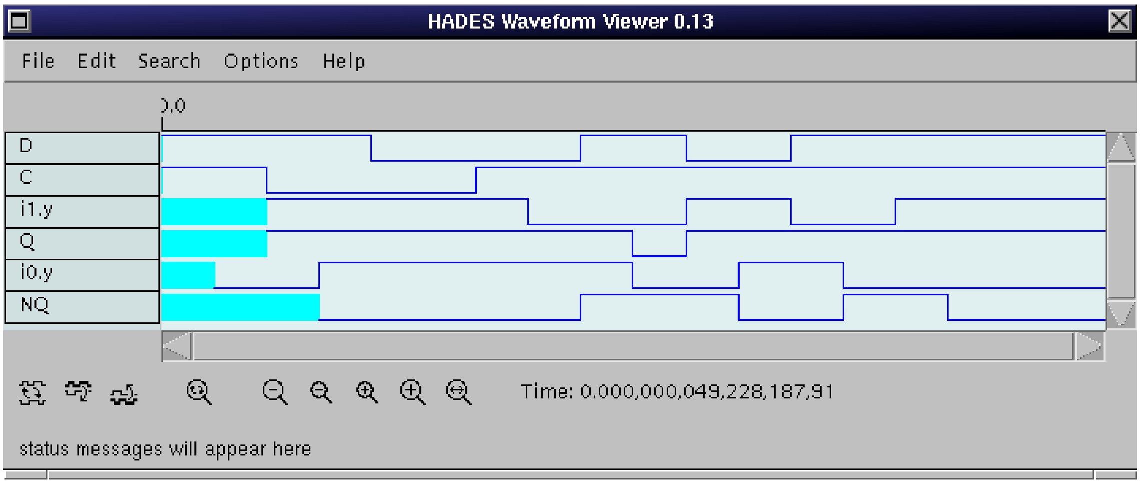

Abbildung 9.2: Der HADES-Waveformviewer zeigt den Signalverlauf eines Simulationsexperimentes in Form eines Impulsdiagrammes an. Designkomponenten befinden sich beim Simulationstart in einem undefinierten Zustand. Die in der STL spezifizierten Stimuli, werden an die Eingänge angelegt und führen zu den dargestellten Signalveränderungen der Signale C und D. Die Schaltung verarbeitet diese Signale und der angegebene Wert des STL-Tests wird mit dem des Q-Signals verglichen.

```
32 v.defStimuli([
33   1, 1, 0,
34   0, 1, ["OR", pD, pC],
35   0, 0, ["OR", pD, pC],
36   1, 0, ["NOT", pD],
37   1, 1, 1,
38   1, 0, ["AND", pC, pD, pI0Y, pI1Y, pNQ, probeQ],
39   1, 1, ["OR", pC, pD, pI0Y, pI1Y, pNQ, probeQ]
40 ])
41
42 # 5. Starten
43 v.start()
```

Die angegebene Stimuliliste (Zeile 32-40) wird durch ein `VectorGen`-Objekt sequentiell abgearbeitet. Dieses erzeugt jeweils für die beiden Eingänge C und D ein Simulationsereignis und ließt, am Ende des Zeitintervalls, den Wert des Abtastpunktes *probeQ* ab. Dessen logischer Wert wird mit dem des booleschen Ausdrucks aus der Stimuliliste verglichen. Das dynamische Verhalten der Simulation läßt sich sehr gut im HADES-Waveformviewer nachvollziehen (s. Abb. 9.2). Die logischen Werte der einzelnen Abtastpunkte werden aufgezeichnet und können durch geeignetes Vergrößern gezielt betrachtet werden.

Kapitel 10

Realisierung der HADES-Komponentenerweiterung

Das im Abschnitt 7.2.2 auf Seite 54 dargestellte Konzept der Komponentenerweiterung von HADES wird in diesem Kapitel an dem Beispiel asynchroner Logik-Komponenten vertieft. Das für das Verständnis benötigte grundlegende Wissen wird im anschließenden Abschnitt vermittelt. Die Entwicklung eines neuen Simulationsobjektes wird am Beispiel eines Muller-C-Gatters erläutert. Wie komplette HADES-Designs mittels Hilfsmodulen erstellt werden können, stellt der Abschnitt 10.3 dar.

Der Schwerpunkt dieses Kapitel liegt bei der Entwicklung neuer HADES-Komponenten. Die hierfür verwendeten asynchronen Komponenten dienen als Beispiel. Die Möglichkeiten dieses Themas wurden nicht ausgeschöpft, sondern eher angerissen (s. Ausblick 11.2 auf Seite 90).

10.1 Grundlagen von asynchronen Komponenten

Die heute übliche Technik der Prozessoren mit zentraler Takteinheit stößt an ihre physikalischen Grenzen [Blu99]. Die zunehmende Chipgröße führt zu immer leistungsstärkeren Takttreibern, die immer längere Signalwege bewältigen müssen. Ihr Platzbedarf steigt und die Verlustleistung der Komponente erhöht sich. Nachteilig ist, daß alle Komponenten im Takt geschaltet werden, unabhängig ob sie etwas getan haben oder nicht. Die Taktfrequenz muß der langsamsten Komponente angepaßt werden. Einen Ausweg aus diesen Problemen könnten die seit den 50er Jahren entwickelten asynchronen Prozessoren darstellen. Sie kommen ohne zentralen Takt aus, die Komponenten vermitteln ihren Zustand über einen Handshake-Mechanismus. Mittels eines `request`-Signals (REQ) meldet ein Logikblock seiner nächsten Stufe, daß neue Daten zur Abnahme bereitstehen. Er selbst nimmt erst wieder ein neues Datum entgegen, wenn seine Daten abgeholt wurden. Diese Zustandsänderung signalisiert der Logikblock seinem Vorgänger mittels eines `acknowledge`-Signals (ACK).

Eine grundlegende Problematik ist, daß die Logikelemente wissen müssten, wann sie fertig sind. Aber woher sollten sie dieses Wissen beziehen? Sie sind aus verschiedenen Gattern aufgebaut, die eine gewisse Schaltzeit erfordern. Diese besitzen unterschiedliche Verarbeitungsgeschwindigkeiten, so können kurzfristig ungültige Bitmuster am Ausgang anliegen.

Der erste Ansatz, der dieses Problem mit normalen Gattern angeht, ist das *bounded*

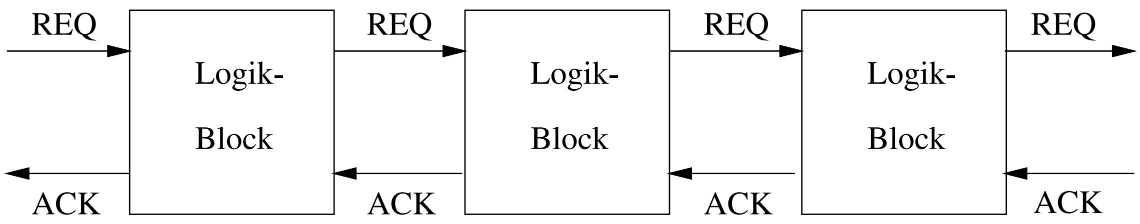

Abbildung 10.1: Logikblöcke tauschen ihre Zustände über eine Handshake-Leitung aus. Auf diese Weise durchlaufen Datenwellen ein asynchrones System. Ein Logikblock benachrichtigt seinen Nachfolger durch ein REQ-Signal, das neue Daten zur Verfügung stehen. Dieser bestätigt deren Abnahme durch ein ACK-Signal.

delay model [LSV93]. Es basiert auf der Annahme, daß Gatter eine maximale Verzögerung niemals überschreiten. Die Grundidee ist, daß ein zusätzliches Verzögerungsgelement nach Erhalten eines REQ-Signals am Eingang genau diese maximale Zeitspanne wartet, und erst dann das ACK-Signal am Ausgang aktiviert.

Ein besserer Ansatz wäre das direkte Messen, ob ein Schaltvorgang beendet ist. Verfahren wie das *Current Sensing Completition Detection*-Verfahren verfolgen diesen Ansatz. Nachteilig ist hierbei, daß zusätzlich zu den digitalen Elementarbausteinen noch analoge Funktionsblöcke auf einem Chip untergebracht werden müssen.

In dieser Arbeit wird der *bounded delay model*-Ansatz verfolgt, bei dem das Design der Logikblöcke identisch zum synchronen System bleiben kann. Lediglich an den Schnittstellen zwischen den Funktionsblöcken tritt die Asynchronität auf.

Die `bounded delay`-Elemente werden mittels einer *Micropipeline* verbunden, die von Ivan Sutherland entworfen wurden [Sus89]. Diese asynchronen FIFO's bestehen aus einem `bounded delay`-Datenpfad und einer Kontroll-Logik. Dieses System kommt ohne Annahmen über Verzögerungszeiten aus und verhält sich robust, da sich die asynchrone Kommunikation unkritisch gegenüber Schaltzeiten verhält.

Das `Muller-C-Gatter`-Element [MB59] ist das Kernstück der Micropipelines. Es verhält sich wie ein spezielles Flipflop, bei dem ein Signal erst dann durchgeschaltet wird, wenn es von beiden Eingängen geliefert wird. Geschieht dies nicht, so behält das Muller-C-Gatter seinen letzten Wert.

10.1.1 Organisationgrundlage der Micropipeline

Das folgende Beispiel [Blu99] verdeutlicht die Arbeitsweise der Muller-C-Gatter und bildet gleichzeitig die Organisationsgrundlage der Micropipelines. Die Angabe der verschiedenen Phasen beziehen sich auf die Abbildung 10.2. Diese Pipeline besteht im wesentlichen aus 3 Muller-C-Gatter und 3 Invertern.

In der ersten Phase wird deren Grundzustand gezeigt. Die beiden Eingänge *REQ_in* und *ACK_in* liefern jeweils eine '0' und die Inverter an ihre Muller-C-Gatter eine '1'. Wird der *REQ_in*-Eingang auf '1' geschaltet, führt dies zur folgenden Schaltsequenz (Phase 2). Beide Eingänge des Muller-C-Gatter C_1 liefern eine '1' woraufhin es schaltet, dadurch kann C_2 ebenfalls auf '1' wechseln und der *ACK_out*-Ausgang wird auf '1' gesetzt. Die Pipeline signalisiert ihrer Vorstufe damit, daß sie deren Daten empfangen hat. C_3 schal-

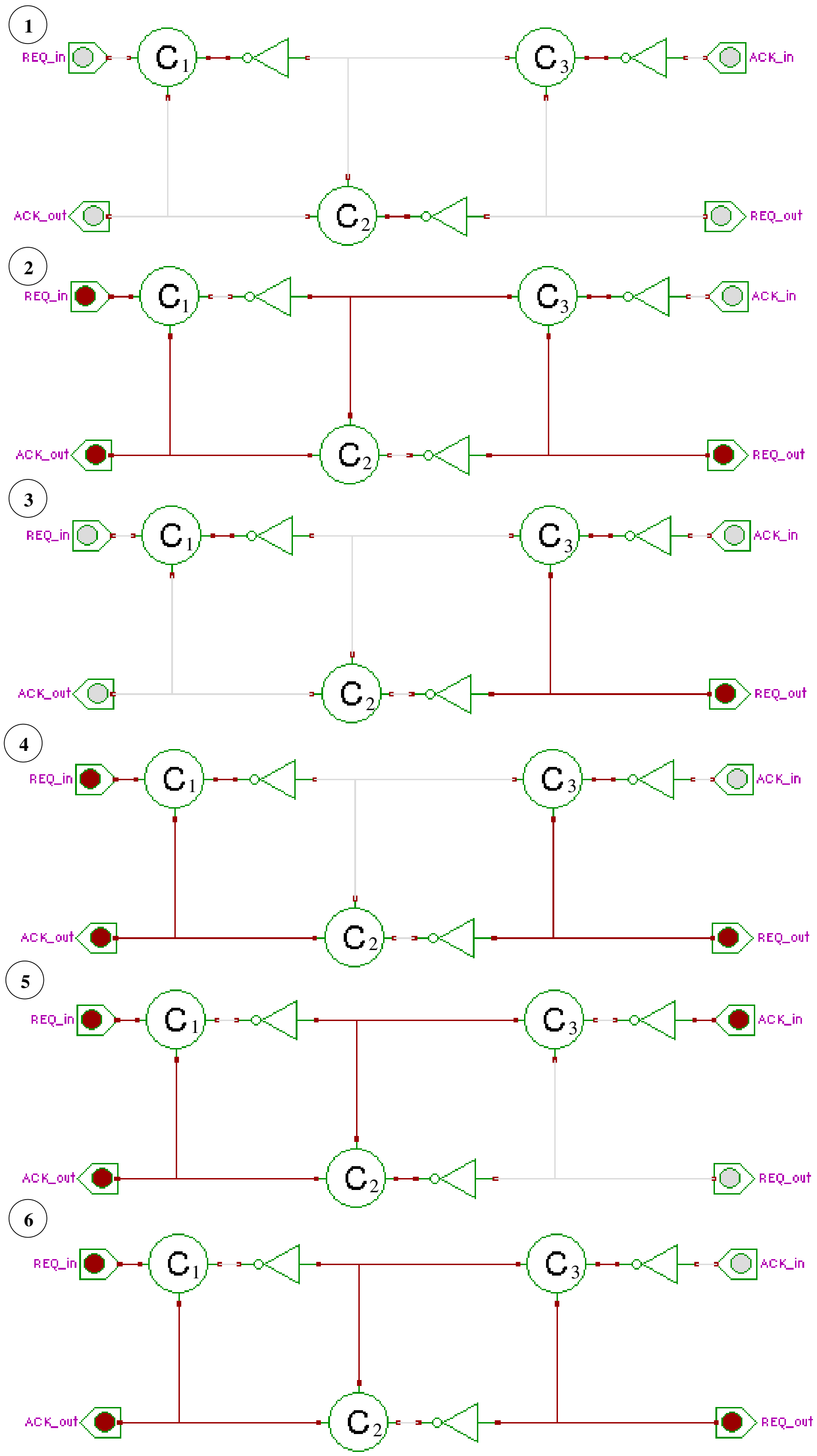

Abbildung 10.2: Die verschiedenen Zustände der Schaltsequenz einer Micropipeline werden dargestellt. Hierdurch werden sowohl die Arbeitsweise des Muller-C-Gatters, als auch die grundlegende Signalorganisation einer Micropipeline verdeutlicht. Die dunklen Leitungen stellen die logische 1 dar.

tet das Signal durch und am *REG_out*-Ausgang erscheint eine '1'. Die Pipeline hat das Signal gespeichert und signalisiert ihrem Nachfolger, daß neue Daten zur Abnahme bereitstehen. Der Vorgänger reagiert auf das *ACK_out*-Signal und zieht sein *REQ_in*-Signal zurück. Hierdurch kann erst C_1 und danach C_2 auf '0' schalten, das Muller-C-Gatter C_3 ist durch ein fehlendes *ACK_in*-Signal des Nachfolgers geblockt (Phase 3). Die Pipeline wartet auf das Abholen ihres *REG_out*-Signals, kann aber ein neues Signal vom Vorgänger entgegennehmen. Dieses geschieht durch ein erneutes *REQ_in*-Signal (Phase 4), C_1 schaltet den *ACK_out*-Ausgang wieder auf '1', aber C_2 kann nicht schalten und das Signal bleibt stecken. C_2 wartet auf ein *ACK_in*-Signal des Nachfolgers. Die Pipeline hat nun 2 Signale gespeichert und kann kein weiteres mehr aufnehmen. Wird *REQ_in* wieder auf '0' gesetzt, kann C_1 dies nicht durchschalten, hierfür müßte C_2 eine '1' liefern und dieses Muller-C-Gatter wartet immer noch auf das *ACK_in*-Signal der nachfolgenden Stufe. In der fünften Phase kommt nun das *ACK_in*-Signal vom Nachfolger, er hat das *REG_out*-Signal übernommen. Sofort schaltet C_3, hierdurch wird der *REG_out*-Ausgang auf '0' gesetzt und C_2 kann den zweiten gespeicherten Impuls weiterschalten. Dieser kann erst durch C_3 an den *ACK_out*-Ausgang geschaltet werden, wenn der Nachfolger neue Daten entgegennehmen kann. Durch eine '0' am *ACK_in*-Eingang zeigt er dieses an (Phase 6) und C_3 leitet den zweiten Impuls an den *REG_out*-Ausgang weiter. Die Pipeline erwartet nun entweder ein erneutes Abholen des Signals durch ihren Nachfolger (*ACK_in*-Signal), oder daß ihr Vorgänger sein *REQ_in*-Signal zurückzieht. Dieses läßt den Signalzyklus wieder von vorne (Phase 1) beginnen. Auf diese Weise organisiert eine Micropipeline ihr Zeitverhalten ohne Taktgenerator.

10.2 Erzeugen neuer Simulationskomponenten

Die HADES-Skriptanbindung beinhaltet die Möglichkeit, neue Simulationsobjekte anstatt in Java durch Python zu realisieren. Dieser Vorgang wird in diesem Abschnitt am Beispiel des Muller-C-Gatters vorgeführt. Hierfür ist zunächst eine Betrachtung der Java-Klasse JPythonWrapper erforderlich, sie ist die Schnittstelle zwischen der Python- und der Java-Welt.

10.2.1 Vorstellung des JPythonWrappers

Jedes Simulationsobjekt in HADES muß als Unterklasse von `SimObject` realisiert sein, so auch die `JPythonWrapper`-Klasse. `SimObject` definiert ein HADES-Basisinterface, dieses wird mit Standardwerten initialisiert. Der JPythonWrapper überschreibt analog zur `GenericGate`-Klasse einige dieser Methoden, um deren Funktionalität für die Python-Module anzupassen. Bei der Entwicklung neuer Simulationsobjekte in Python sind nur wenige Methoden zu realisieren, da ein Großteil vom JPythonWrapper übernommen wird. Dieser enthält äquivalent zur `JPShell`-Klasse ein Pythoninterpreter-Objekt, womit die entsprechenden Python-Simulationsobjekte erzeugt werden. In der folgenden Übersicht werden die wichtigsten Methoden in ihrer Funktionsweise dargestellt.

- Konstruktor
  Er besteht nur aus dem Aufruf des Standardkonstruktors der `SimObject`-Superklasse. Dieser legt ein Array vom Typ `Port` für die Ein- und Ausgänge der Komponente an.

- getSymbolResourceName()
  Diese Methode reicht den Aufruf an das interne Python-Objekt weiter, dieses liefert den Namen des Symbols.

- write()
  Wird ein Simulationsobjekt in einer HADES-Datei gespeichert, sorgt diese Methode dafür, daß alle relevanten Daten hinterlegt werden. Der JPythonWrapper speichert zusätzlich das Python-Modul, die Python-Klasse und die in einer entsprechenden Variablen gehalten Argumente des internen Python-Objektes.

- initialize()
  Der übergebene Konfigurations-String stammt aus der HADES-Datei, die durch die `write`-Methode erzeugt wurde. Mit seinen Argumenten wird ein neues Python-Objekt erzeugt und dessen Referenz als internes Python-Objekt gehalten. Das Erzeugen findet in der folgenden `createJPObject`-Methode statt.

- createJPObject()
  Diese Hilfsmethode beinhaltet den wichtigsten Teil der Komponentenerweiterung durch Python-Module und wird deshalb im Quelltext diskutiert. Dieser ist auf das Wesentliche gekürzt. Dargestellt wird die Erzeugung eines neuen Python-Objektes innerhalb des Interpreters. Anschließend wird die Referenz dieses Objektes an Java übergeben und innerhalb des JPythonWrappers gehalten.

  ```
  public JPythonWrapper createJPObject(String pyModul,
                                       String pyClass,
                                       String args)
  {
      interp.exec("from "+pyModul+" import "+pyClass);
      interp.exec("pyObj = "+pyClass+"()");

      if (args.length() > 0) {
        interp.exec("pyObj.setArgs('"+args+"')");
      }

      Class javaClass = new JPythonWrapper().getClass();
      pyObj = (JPythonWrapper) interp.get("pyObj", javaClass);

      return pyObj;
  }
  ```

  Nachdem die Python-Klasse aus dem spezifizierten Python-Modul importiert wurde, kann ein neues Python-Objekt innerhalb des Interpreters erzeugt werden. Die gegebenenfalls in der Konfigurationsdatei gespeicherten Argumente werden an das neue Python-Objekt weiter gegeben. Anschließend wird dieses aus dem Interpreter geholt und zu einem JPythonWrapper-Objekt gewandelt und zurückgegeben.

- elaborate()
  Beim Simulationsstart wird diese Methode vom Simulator einmal aufgerufen, und sie

wird zur Initialisierung der Komponente verwendet. Sie wird an das interne Python-Objekt weitergegeben.

- evaluate()
  Durch ein Simulationsereignis an einem Port der Komponente wird diese Methode aufgerufen. In ihr wird das Verhalten des Simulationsobjektes festgelegt, daher wird sie an das interne Python-Objekt weitergeleitet.

- configure()
  Dem Benutzer wird durch diese Methode die Manipulation des Objektes während der Simulation durch einen Dialog ermöglicht.

- copy()
  Unter Verwendung der `createJPObject`-Methode wird ein neues Python-Objekt erzeugt und initialisiert. Dieses wird ein neuens JPythonWrapper-Exemplar als internes Python-Objekt übergeben.

- scheduleEventAfter()
  Aus den erhaltenen Argumenten wird ein neues Simulationsereignis erzeugt und dem Simulator übergeben. Diese Methode wird aus der `evaluate`-Methode der Python-Klassen aufgerufen.

Der innerhalb des JPythonWrappers verwendete Interpreter wird möglichst von einer bestehenden jpshell geholt. Sollte diese Möglichkeit nicht bestehen, so wird ein eigener Interpreter erzeugt und initialisiert. Dieses kann zum Beispiel beim Laden eines gespeicherten Designs geschehen.

10.2.2 Beispiel: Muller-C-Gatter

Das Muller-C-Schaltgatter ist das Basiselement der Micropipelines. Es hat zwei Eingänge A und B, sowie einen Ausgang Y. Dieser wird auf logisch 1 geschaltet, wenn an beiden Eingängen jeweils eine logische 1 anliegt. Liegt an den Eingängen eine 0 an, so schaltet der Ausgang auch auf 0. Bei unterschiedlichen Werten der Eingänge behält das Muller-C-Gatter seinen vorherigen Wert bei. Das Muller-C-Gatter ähnelt einem Flipflop und es läßt sich auch aus zwei UND-Gattern und einem RS-Flipflop realisieren (s. Abb. 10.3).

Implementierung in Python

Anhand des Python Quelltextes des Muller-C-Gatters wird auf die Implementierung neuer Komponenten eingegangen. Die Python `Muller`-Klasse wird dazu in einzelne Abschnitte aufgeteilt und nacheinander erklärt. Wie für Python-Klassen üblich, beginnt das Modul mit einem Block von unterschiedlichen `import`-Anweisungen. Importiert wird das normale Python `string`-Modul (Zeile 1), es definiert Operationen für den `string`-Type von Python. In Zeile 2 wird die JPython `jarray`-Klasse aus dem JPython `array`-Modul importiert, sie entspricht der Java `Array`-Klasse. Diese können in Python angelegt, manipuliert und an Java-Klassen übergeben werden. Hiernach wird das `jp_sim_control`-Modul 8.1.1 auf Seite 60 der Skriptanbindung importiert. Die Zeilen 4-6 importieren Java-Klassen aus dem HADES-Framework, die für die Integration und Implementierung einer neuen Simulationskomponente benötigt werden.

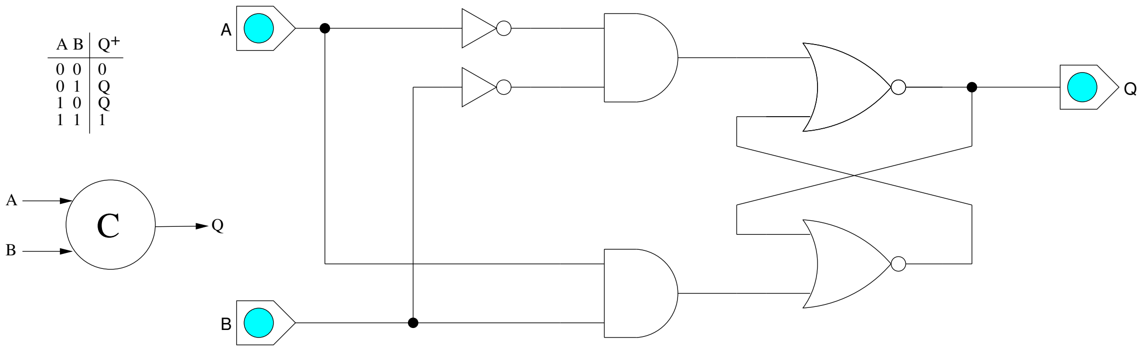

Abbildung 10.3: Kernstück der Micropipelines ist das Muller-C-Gatter, daß sich durch ein RS-Flipflop und einige Gatter realisieren läßt. Liefern beide Eingänge denselben Wert, wird dieser durchgeschaltet; geschieht dies nicht, so behält das Muller-C-Gatter seinen alten Wert.

```
 1 import string
 2 from jarray import array
 3 import jp_sim_control
 4 from hades.jpshell import JPythonWrapper
 5 from hades.models import PortStdLogic1164, StdLogic1164
 6 from hades.simulator import Port
```

Der Import-Block wird gefolgt von der Klassendefinition (Zeile 7), sie deklariert eine `Muller`-Klasse als Subklasse von der Java `JPythonWrapper`-Klasse. Der Konstruktor (Zeile 9-20) beginnt mit einem Konstruktoraufruf der Superklasse. Hierdurch werden die benötigten `Port`-Variablen und Array angelegt und können anschließend von der Komponente gesetzt werden. Dieses Füllen mit den richtigen Werten wird durch das Erzeugen der `Port`-Objekte (Zeile 12-14) realisiert. Die explizite Angabe der `self`-Referenz ist durch die Python-Syntax vorgeschrieben und entspricht der aus Java bekannten `this`-Referenz. In der Zeile 15 werden die Ports des Objektes in einem Array gruppiert und das wichtige `ports`-Array der Superklasse gesetzt. Die Schaltzeit der Komponente wird in der `t_delay`-Variable gespeichert. Hiernach folgt der Muller-spezifische-Variablen-Teil, so wird das horizontale Symbol als Standard gesetzt.Durch die Zeile 21 wird der Symboltyp mit in die Konfigurationsdatei geschrieben.

```
 7 class Muller (JPythonWrapper):
 8
 9     def __init__(self):
10         JPythonWrapper.__init__(self)
11
12         self.port_Y = PortStdLogic1164( self, "Y", Port.OUT, None )
13         self.port_A = PortStdLogic1164( self, "A", Port.IN, None )
14         self.port_B = PortStdLogic1164( self, "B", Port.IN, None )
15         self.ports = array([self.port_A, self.port_B, self.port_Y], Port)
16
```

```
17          self.t_delay = 5.0E-9
18          self.next_Y = StdLogic1164( StdLogic1164._U )
19          self.old_value = StdLogic1164( StdLogic1164._U )
20          self.symbol_type = "horizontal"
21          self.pyWriteArgs = self.symbol_type
```

Der HADES-Simulator ruft beim Start einer Simulation die `elaborate`-Methode der Komponenten auf. Dieser Initialisierungschritt nutzt die Muller-Klasse um sich eine Referenz auf den Simulator zu holen und damit die geerbte Referenz von der `SimObject`-Klasse zu aktualisieren.

```
22      def elaborate(self, arg):
23          self.simulator = jp_sim_contrrol.getSimulator()
```

Die `evaluate`-Methode spezifiziert das Simulationsverhalten der Komponente. Für die Implementierung des Muller-C-Gatters bedeutet dies, daß geprüft werden muß, ob die beiden Eingänge identische Signale liefern (Zeile 30). Wenn dies der Fall ist, wird der neue Wert für den Ausgang durch eine UND-Verknüpfung ermittelt, sonst wird der alte Wert beibehalten. Schließlich wird durch den Aufruf der `scheduleEventAfter`-Methode aus der JPythonWrapper-Klasse ein neues Simulationsereignis dem Simulator übergeben.

```
24      def evaluate(self, arg):
25          signal_Y = self.port_Y.getSignal()
26
27          value_A = self.port_A.getValueOrU()
28          value_B = self.port_B.getValueOrU()
29
30          if value_A == value_B:
31              self.next_Y.setToAnd( value_A, value_B )
32              self.old_value = self.next_Y
33          else:
34              self.next_Y=self.old_value
35
36          JPythonWrapper.scheduleEventAfter( self, signal_Y, self.t_delay,
37                                             self.next_Y, self.port_Y )
```

Für die graphische Darstellung durch den HADES-Editor, muß dieser die Symboldatei der Komponente ermitteln. Dies wird durch die `getSymbolResourceName`-Methode realisiert. Das Muller-C-Gatter liefert abhängig von deren Typ verschiedene Symbole zurück (s. Abb. 10.4)

```
38      def getSymbolResourceName(self):
39          if string.upper(self.symbol_type) == "SOUTH":
40              return "/hades/jpshell/module/objects/MullerC_V_S.sym"
41          elif string.upper(self.symbol_type) == "NORTH":
42              return "/hades/jpshell/module/objects/MullerC_V_N.sym"
43          else:
44              return "/hades/jpshell/module/objects/MullerC_H.sym"
```

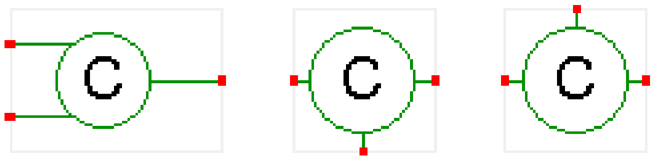

Abbildung 10.4: Für die Simulationskomponente des Muller-C-Gatter existieren verschiedene HADES-Symboltypen; die horizontale und die vertikale mit einem Süd- oder Nord-Ausgang.

10.3 Erzeugen neuer HADES-Designs

Nachdem die Implementierung neuer HADES-Komponenten anhand des Muller-C-Gatters ausführlich dargestellt wurde, wird jetzt auf die Erzeugung kompletter HADES-Designs eingegangen. Hierfür wurde bewußt ein sehr einfaches HADES-Design gewählt (s. Abb. 10.5), dieses umfaßt jedoch alle Mechanismen. Auch komplexere Designs, wie z.B. eine komplette Micropipeline (s. Abb. 10.6), können komplett durch ein Modul erzeugt werden.

Ein einfaches Muller Design

Das folgende kurze Python Moduls erzeugt ein einfaches HADES-Design. Es besteht aus vier Blöcken, der erste importiert die benötigten Module und Klassen. Der zweite (Zeile 6-9) legt die Positionen der Komponenten fest, in diesem Beispiel sind diese Punkte die zentralen Daten für die Erzeugung des Designs. Der dritte Block erzeugt die Komponenten an der spezifizierten Position im Editor und ihr Name innerhalb des Designs wird angegeben. Beendet wird dieses Modul durch das Verbinden der Komponenten.

```
1 from jp_design_os import *
2 from Muller import Muller
3 from hades.models.io import Ipin, Opin
4 from java.awt import Point
5
6 muller_pos  = Point(6000,3600)    // Vielfaches von 600, bei 2400 dpi
7 a_pos       = Point(4800,2400)
8 b_pos       = Point(4800,7200)
9 y_pos       = Point(10800,4800)
10
11 addComponentWithSymbol(Muller(), muller_pos,   "muller")
12 addComponentWithSymbol(Ipin(),   a_pos,        "A")
13 addComponentWithSymbol(Ipin(),   b_pos,        "B")
14 addComponentWithSymbol(Opin(),   y_pos,        "Y")
15
16 connectWithSignalPos(a_pos,      "Y", muller_pos, "A")
17 connectWithSignalPos(b_pos,      "Y", muller_pos, "B")
18 connectWithSignalPos(muller_pos, "Y", y_pos,      "A")
```

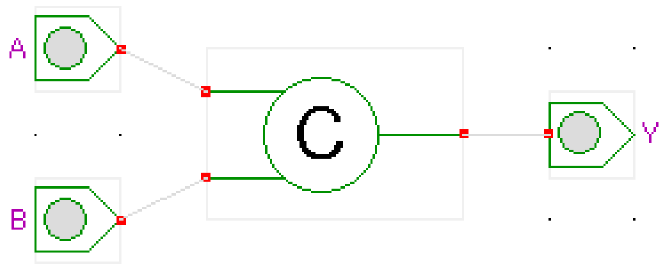

Abbildung 10.5: Dieses einfache HADES-Design mit einem Muller-C-Gatter wurde durch ein kleines JPython-Modul erzeugt (s. Seite 83).

Dieses Design wird durch eine `import`-Anweisung aus der jpshell erzeugt und kann wie jedes Design manipuliert und gesichert werden. Ein zusätzlicher STL-Teil kann beliebig in das Modul aufgenommen werden.

10.3.1 Beispiel: Micropipeline

Komplexere Designs können ebenfalls durch Module der HADES-Skriptanbindung erzeugt werden, z.B. eine komplette Micropipeline.

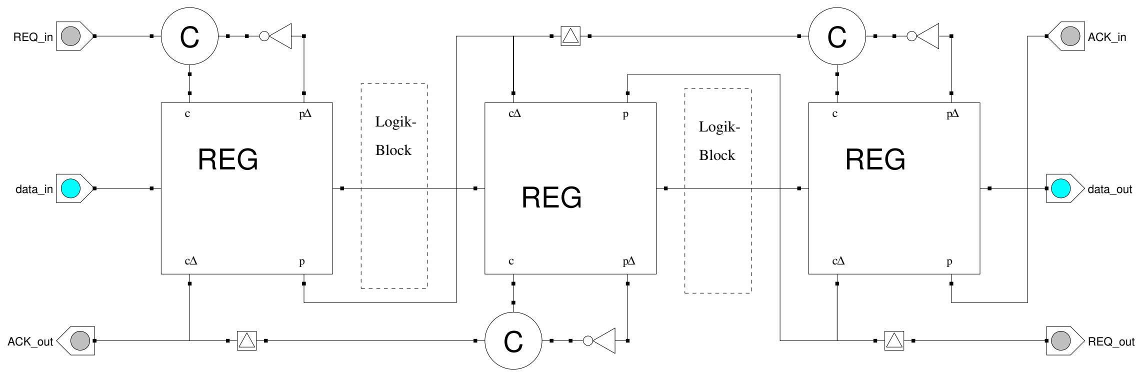

Abbildung 10.6: Das HADES-Design der Micropipeline ist mit Ausnahme der Inverter komplett in JPython implementiert. Die einzelnen Komponenten wie das Muller-C-Gatter, die speziellen Register sowie die Verzögerungselemente sind als Python-Klassen realisiert. Das dargestellte System arbeitet asynchron und organisiert seine Signalläufe selbst.

Aus Muller-C-Gattern und Invertern läßt sich ein FIFO-Speicher für Impulse aufbauen (s. Abschnitt 10.2).

Eine Micropipeline kann erst Daten berechnen, wenn sie um Register, Verzögerungsgatter und Logik erweitert wird. Diese Register und die Verzögerungsgatter sind so in die Muller-C-Gatter Pipeline einzubauen, daß die Reihenfolge der Impulse in der Pipeline genau dem Ablauf der Berechnung entspricht.

Jedes Register besitzt zwei Takteingänge c (Capture) und p (Pass), sowie zwei Taktausgänge $c\Delta$ und $p\Delta$. Das Register nimmt die Datenwerte des *data_in*-Eingangs durch ein entsprechendes Signal am c-Eingang entgegen und speichert es. Mittels eines Signals am p-Eingang erscheint dieser gespeicherte Wert am *data_out*-Ausgang des Registers. Die eingegangenen Taksignale werden erst nach einer Verzögerung an die entsprechenden Taktausgänge weiter gegeben. Hierdurch wird sichergestellt, daß das Register seine Arbeit beendet hat. Die Verzögerungsgatter müssen auf die maximale Verarbeitungszeit der synchronen Logikblöcke ausgelegt sein und garantieren somit deren korrektes Verhalten. Durch die Hinzunahme dieser Komponenten erhält die Micropipeline ihr *bounded-delay*-Verhalten uns organisiert ihr asynchrones Zeitverhalten selbst.

Teil IV

Resümee

Kapitel 11

Bewertung und Ausblick

Dieses Kapitel blickt auf die Arbeit zurück. Deren Resultate werden vor dem Hintergrund der Aufgabenstellung bewertet. In einem Ausblick werden zukünftige HADES-Systemerweiterungen angedacht, sowie neuste Veränderungen der Skriptsprachen dargestellt.

11.1 Abschließende Bewertung der Skriptanbindung

Ziel dieser Arbeit war es, das HADES-Framework um eine Skriptsprache zu erweitern. Diese Skriptanbindung sollte vollen Zugriff auf die bestehende Java-Applikation bieten und die interaktive Manipulation von Simulationsexperimenten ermöglichen. Die bestehenden Grenzen der Benutzeroberfläche von HADES waren durch eine kommandozeilenorientierte Shell zu Überwinden. Aufbauend auf diese Basisfunktionalität sollte HADES mit einer Simulations- und Test-Sprache versehen werden, die das Treiben und Testen bestehender HADES-Designs ermöglichen muß. Die Skriptanbindung sollte die Entwicklung neuer Simulationskomponenten zur Laufzeit unterstützen. Um diese praktischen Problemstellung zu realisieren, ist eine vollwertige Skriptsprache mit einer Anbindung von Java nötig. Bestehende Skriptsprachen waren auf ihre Tauglichkeit für ihren Einsatz in dieser Arbeit hin zu untersuchen und gegebenenfalls eine auszuwählen.

Das Gesamtziel der Realisierung einer HADES-Skriptanbindung ist erfüllt worden. Im theoretischen Teil wurde mittels einer Evaluation bestehender Skriptsprachen und ihrer Java-Interpreter, JPython als Skriptsprache ausgewählt. Rückblickend spiegeln die untersuchten Interpreter das vorhandene Spektrum wieder. Die einzelnen Projekte haben ihre eigenen Vor- und Nachteile und sind vom Funktionsumfang ähnlich wie Jacl und Skij. Andere befinden sich, wie FESI in einem unvollendeten Zustand, deren Dokumentation ist meistens unzureichend. Die Auswahl von FESI als JavaScript-Interpreter muß rückwirkend als falsch angesehen werden. Mit Rhino [Moz2000] exisitert ein ausgereifter und sehr gut dokumentierter Interpreter für EcmaScript, dieser ist FESI vorzuziehen.

Für spezielle Aufgabenstellungen existieren kleine Spezialisten wie die Beanshell, aber JPython nimmt deutlich eine Führungsposition ein. JPython überzeugt durch seine elegante Syntax und die gute technische Umsetzung. Die zugrundeliegende Skriptsprache Python ist funktional mit Java gleichzusetzen. JPython vereint die beiden Sprachen Java und Python zu einem mächtigen Sprachgemisch.

Aus dem Ansatz von JPython, die Module in Java-Bytecode zu übersetzen und dann durch die Java-VM interpretieren zu lassen, resultiert die enorme Geschwindigkeit. Insgesamt kann festgestellt werden, daß die einzelnen praktischen Teilaufgaben mit JPython umfassend und ohne Einschränkungen realisiert werden konnten.

Die Skriptanbindung ist gradlinig und ermöglicht die Einhaltung der konzeptionellen Ziele. Dieses wird, z.B. deutlich bei den Benutzer-Modulen der Komponentenerweiterung. Ihr Quelltext ist kürzer als der ihrer Java-Varianten, obwohl sie exakt dieselbe Funktionalität bieten. Ihr Aufbau ist ebenfalls gleich und sie können direkt interaktiv in der Shell der Skriptanbindung getestet werden. Ein übersetzen der JPython-Klassen entfällt, wodurch sich der Entwicklungszyklus verkürzt. Dieses Beispiel belegt die Vorteile von Skriptsprachen gegenüber Systemprogrammiersprachen. Bei der Realisierung der Komponentenerweiterung von HADES, sind technische Unterschiede in der Objekt Erzeugung zwischen Java und JPython aufgetreten. Diese konnten aber durch das Einfügen einer JPythonWrapper-Klasse aufgehoben werden. Weitere Unzulänglichkeiten von JPython sind in der Realisierung nicht aufgetreten.

11.2 Ausblick

Im Rahmen dieser Arbeit wurden das Thema der asynchronen Komponenten als Beispielimplementation der HADES-Komponentenerweiterung verwendet. Der Schwerpunkt hierbei lag in der Realisierung von Simulationsobjekten in JPython. Hierfür wurden das *bounded delay*-Modell gewählt und eine asynchrone Micropipeline in ihren Grundzügen implementiert. Andere asynchrone Ansätze wurden nicht verfolgt und auch die Möglichkeiten der Micropipeline nicht ausgeschöpft. In diesem interessanten Themenkomplex sollte, aufbauend auf die in dieser Arbeit erarbeiteten Grundlagen, eine komplette asynchrone HADES-Komponentenbibliothek entstehen. Hierdurch wäre HADES in der Lage, Designs mit asynchronen Signalleitungen zu simulieren und erforschen.

Der theoretische Teil dieser Arbeit beleuchtet den Stand der Skriptsprachen und deren Java-Anbindungen Ende 1999. Bei den Skriptsprachen ist gerade in den letzten Monaten eine deutlich zunehmende Bewegung zu Java festzustellen. Projekte wie z.B. Jacl/Blend werden jetzt, nach einer langen Entwicklungspause, wieder weiter vorangetrieben. Eine umfassende und aktuelle Übersicht aller Projekte die sich mit und um die Java-VM befassen ist unter [Tol2000] zu finden.

Teil V

Anhänge

Plattformtest von JPython

JPython in der Release-Version 1.1 wurde auf den Plattformen Linux (Kernel 2.2.13), Windows98 und Solaris 2.6 getestet. JPython lief auf jeder Plattform und unter den verschiedenen JDKs stabil und fehlerfrei.

Plattform	Java-Version	Hersteller	JPython-Version	Stabilität	Performance
Linux 2.2.13	jdk1.1.8	IBM	JPython 1.1	Ok	ev/sec=150000
Linux 2.2.13	jdk1.2.2	Blackdown	JPython 1.1	Ok	ev/sec=80000
Windows98 II	sdk2 v1.2.2-001	Sun	JPython 1.1	Ok	ev/sec=210000
Windows98 II	hotspot v1.0.1	Sun	JPython 1.1	Ok	ev/sec=470000
Windows98 II	hotspot v2	Sun	JPython 1.1	Ok	ev/sec=650000

Tabelle 1: Plattformtest von JPython

Die hierbei gemessene HADES-Performance (events/second) ist sehr unterschiedlich. Das beste Ergebnis wurde mit der `hotspot2`-Architektur unter Windows98 erzielt (Test-Rechner: Athlon 500, 128MB, Matrox G400).

Literaturverzeichnis

[ASS91] H. Abelson, G.J. Sussmann, J. Sussmann, *Struktur und Interpretation von Computerprogrammen*, Springer-Verlag, 1991

[Bal99] S. Balmurugan, *Jperl: Accessing Perl from Java*, Dr. Dobb's Journal, Februar 1999

[Blu99] Andreas Bleul, *Die Rückkehr asynchroner Prozessoren*, c't, Heft 17, 1999

[Bot98] Per Bothner, *Kawa: Compiling Scheme to Java*, `http://source-ware.cygnus.com/kawa/papers/KawaLisp98.ps.gz`, 1998

[Cad97] *Design Framework II 4.41 User Manual*, Cadence Design Systems, Inc., San Jose, CA, USA, 1997

[Cas98] Luke Andrew Cassady-Dorian, *Scripting tools and java*, JavaPro, `http://www.devx.com/upload/free/features/javapro/1998/12-dec98/ld1298/ld1298.asp`, Dezember, 1998

[Chan98] Patrick Chan, *The java developers almanac*, Addison-Wesley, 1998

[CR98] W. Clinger, J. Rees u.a, *Revised(4) Report on The Algorithmic Language Scheme*, `http://www.angrygraycat.com/scheme/doc/r4rs.htm`, 1998

[DD96] R. Kent Dybvig, Kent Dybbig, *Scheme Programming Language, The: ANSI Scheme*, Prentice Hall, 1996

[DL99] Moses DeJong, Cameron Laird, *Tcl + Java = A match made for scripting*, `http://www.sunworld.com/swol-11-1999/swol-11-jacl.html`, 1999

[ECMA99] *ECMA-262 ECMAScript Language Specification, 3rd edition*, `ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf`, 1999

[Fla97] David Flanagan, *JavaScript: The definitve guide*, O'Reilly & Associates, 1997

[Fla98] David Flanagan, *Java in a nutshell*, O'Reilly & Associates, 1998

[FSF] Free Software Foundation, `http://www.gnu.org/fsf/fsf.html`, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA

[GJS96] J. Gosling, B. Joy, G. Steele, *The Java[tm] Language Specification*, Addison-Wesley, 1996

[Har99] Greg Harvey, *Quick and Dirty Guile Docs*, `http://home.the-zone.net/~gharvey/guile/qdocs/html/qdocs_toc.html`, 1999

[HC99] Cay S. Horstmann, Gary Cornell, *core java 1.2, volume I-fundamentals*, Sun Microsystems Press, 1999

[Hen97] N. Hendrich, *HADES Tutorial 0.43*, Universität Hamburg, FB Informatik, 1997

[Hen97b] N. Hendrich, *Java für Fortgeschrittene*, Springer-Verlag, 1997

[Hen98] N. Hendrich, *HADES homepage*, Universität Hamburg, FB Informatik, `tech-www.informatik.uni-hamburg.de/applets/hades`, 1998

[Hen2000a] N. Hendrich, M. Grove, *Praktikum Technische Informatik T3*, Universität Hamburg, FB Informatik, `http://tech-www.informatik.uni-hamburg.de/Dokumentation/epra/t3-1.ps.gz`, 2000

[Hen2000b] N. Hendrich, *A Java-based framework for simulation and teaching*, Proc. EWME 2000, Aix-en-Provence, to appear

[HM99] Tobias Himsted, Klaus Mätzel, *Mit Python programmieren: Einführung und Anwendung skriptorientierter Programmierung*, dpunkt, 1999

[Hua98] Samuel F. Huang, *Java reflection and java beans*, `http://sern.ucalgary.ca/Courses/CPSC/547/F98/Slides/Huang/javabeans.htm`, 1998

[Hug97] J. Hugunin, *Python and Java: the Best of both worlds*, Proc. 6th International Python Conference, San Jose, California, `http://www.python.org/workshops/1997-10/proceedings/hugunin.html`, 1997

[Hug97b] J. Hugunin, *JPython tutorial*, Proc. 6th International Python Conference, San Jose, California, 1997

[IEEE-87] IEEE Standard VHDL, *Hardware Desciption Language Reference Manual*, IEEE Std 1076-1987, 1987

[Jaw99] J. Jaworski, *Mastering JavaScript and JScript, The Essential Reference for Building Interactive Web Pages. Covers JavaScript 1.3 and JScript 5*, Sybex Inc., 1999

[Joh98] Ray Johnson, *Tcl and Java Integration*, Sun Microsystems, `http://sunscript.sun.com/java/tcljava.ps`, 1998

[KR88] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice Hall, 1988

[Lad99] Ramnivas Laddad, *Scripting power saves the day for your java apps*, JavaWorld http://www.javaworld.com/jw-10-1999/jw-10-script.html, 1999

[Lag87] Prof. Dr.-Ing. K. Lagemann, *Rechnerstrukturen*, Springer-Verlag, 1987

[LSV93] L. Lavagno, A. Sangiovanni-Vincentelli, *Algorithms for Systhesis and Testing of Asynchronous Circuits*, Kluwer Academic, 1993

[LS97] Cameron Laird, Kathryn Soraiz, *Choosing a scripting language*, SunWorld, Oktober 1999

[Lug2000] JeanMarc Lugrin, *FESI a free EcmaScript Interpreter, Release 1.1.4*, `http://home.worldcom.ch/jmlugrin/fesi/index.html`, 2000

[Lut96] Mark Lutz, *Programming Python*, O'Reilly & Associates, 1996

[Lut99] Mark Lutz, *Python kurz & gut*, O'Reilly & Associates, 1999

[LY97] Tim Lindholm, Frank Yellin, *The java virtual machine specification*, Addison-Wesley, 1997

[Mas96] Roger E. Masse, *Develepment Strategies for Better Software: An Analysis of Two NextGeneration Languages, Java and Python*, `http://www.python.org/~rmasse/papers/java-python/`, 1996

[MB59] D. E. Muller, W. C. Bartky, *A theory of asynchronous cicuits*, Annals of Computing Laboratory of Harvard University, page 204-243, 1959

[McC2000] John H. McCoy, *Scripting for Pnuts*, Dr. Dobb's Journal, Januar 2000

[McM97] Chuck McManis, *Take an in-depth look at the java reflection API*, JavaWorld `http://www.javaworld.com/jw-09-1997/jw-09-indepth.html`, 1997

[MD97] Jon Meyer, Troy Downing, *Java Virtual Machine*, O'Reilly & Associates, 1997

[Meyer98] M. Meyer, *Konzeption und Umsetzung eines Java-Applets zur Logikminimierung mit KV-Diagrammen*, Studienarbeit, Fachbereich Informatik, Universität Hamburg, `http://tech-www.informatik.uni-hamburg.de/applets/kvd/`, 1998

[MH2000] R. Monson-Haefel, *Enterprise JavaBeans*, O'Reilly & Associates, 2000

[MS95] *Microsoft Visual Basic Programmers Guide*, Microsoft Corporation, U.S., 1995

[Moz2000] The Mozilla Organization, *Rhino Documentation*, `http://www.mozilla.org/rhino/doc.html`, 2000

[Niem2000] Pat Niemeyer, *Beanshell: The User Manual*, `http://www.beanshell.org/manual/contents.html`, 2000

[Niem2000b] Pat Niemeyer, *BeanShell Commands*, `http://www.beanshell.org/manual/bshcommands.html`, 2000

[Ous94] John K. Ousterhout, *Tcl and the Tk toolkit*, Addidon-Wesley, 1994

[Ous98] John K. Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, IEEE Computer magazine, März 1998

[Page91] B. Page, *Diskrete Simulation*, Springer Verlag, Berlin, 1991

[Rink2000] Claas Rink, *Simulation eines MIPS-R3000-kompatiblen Mikroprozessors als HADES-Modell*, Diplomarbeit, Fachbereich Informatik, Universität Hamburg, `http://tech-www.informatik.uni-hamburg.de/applets/hades/`, 2000

[Ros95] Guido van Rossum, *Python Library Reference*, CWI Report CS-R9524, 1995

[Ros95b] Guido van Rossum, *Python reference manual*, CWI Report CS-R9525, 1995

[Ros95c] Guido van Rossum, *Python tutorial*, CWI Report CS-R9526, 1995

[Ros95d] Guido van Rossum, *Extending and embedding the Python interpreter*, CWI Report CS-R9527, 1995

[Ros98] Guido van Rossum, *Java and Python: a perfect couple*, `http://developer.earthweb.com/journal/techfocus/081798_jpython.html`, 1998

[Ros98b] Guido van Rossum, *Glue it all together with python*, `http://www.python.org/doc/essays/omg-darpa-mcc-position.html`, 1998

[Ros98c] Guido van Rossum, *Comparing python to other Languages*, `http://www.python.org/doc/essays/comparisons.html`, 1998

[Ros2000] Guido van Rossum, *Python reference manual, Release 1.5.2*, `http://www.python.org/ftp/python/doc/1.5.2p2/postscript-a4--1.5.2p2.tgz`, 2000

[Sch84] A. Schulz, *Einführung in das Programmieren in PL/1*, de Gruyter, 1984

[Scr2000] Scriptics, Inc., Tcl-Webserver, `http://dev.scriptics.com/software/tcl-httpd`, 2000

[SS92] W.Schiffmann, R. Schmitz, *Technische Informatik 1: Grundlgen der digitalen Elektronik*, Springer-Verlag, 1992

[Sta98] Scott Stanton, *TclBlend: Blending Tcl and Java*, Dr. Dobb's Journal, Februar 1998

[Str92] B. Stroustrup, *Die C++ Programmiersprache*, Addison-Wesley, 1992

[Sun96] Sun Microsystems, *Java Beans Specification, version 1.0*, Sun Microsystems, Mountain View, CA., `http://java.sun.com/beans`, 1996

[Sun97] Sun Microsystems, *Sun 100% pure Java initiative*, Sun Microsystems, Mountain View, CA., `http://java.sun.com/purejava`, 1997

[Sun97b] Sun Microsystems, *Java Core Reflection, API and Specification*, Sun Microsystems, Mountain View, CA., `http://java.sun.com/products//jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html`, 1997

[Sun99] Sun Microsystems, *The Java Hotspot performance engine*, Sun Microelectronics, Mountain View, CA., `http://www.javasoft.com/products/hotspot`, 1999

[Sun2000] Sun Microsystems, *Enterprise Java Beans Specification, Version 2.0*, Sun Microelectronics, Mountain View, CA., `http://java.sun.com/products/ejb/docs.html`, 2000

[Sus89] Ivan E. Sutherland, *Micropipelines*, Communications of the ACM, Volume 32, Nummer 6, Juni 1989

[Tra98] Michael Travers, *Skij*, `http://www.alphaworks.ibm.com/tech/Skij`, 1998

[Tra2000] Michael Travers, *What is interactive Scripting?*, Dr. Dobb's Journal, January 2000

[Tol2000] Robert Tolksdorf, *Programming Languages for the java virtual machine*, `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`, 2000

[WCSP96] Larry Wall, Tom Christiansen, Randal L. Schwartz, Stephen Potter, *Programming Perl (2nd Edition)*, O'Reilly & Associates, 1996