Visualisierung von π-Programmen

Informatik Projekt

von
Cristina Gheorghiu Cris

Betreuer: Franz Achermann
Software Composition Group

Institut für Informatik und angewandte Mathematik
Universität Bern, Neubrückstr. 10, Bern

Januar 1999

Zusammenfassung

Die Visualisierung von Programmen erlaubt, durch animiertes Anzeigen, die Darstellung bedeutender Zustände, die während der Programmausführung auftreten können und sonst unsichtbar für den Benutzer bleiben.

Ziel dieser Arbeit ist die Implementation eines interaktiven Visualisierungswerkzeuges für das JPict Framework.

In dieser Dokumentation werden Konzepte des JPict und HotDraw Frameworks und der Visualisierungsmodelle kurz dargestellt. Danach wird auf das Design und den verfolgten Ansatz in der Implementierung dieses Visualisierungswerkzeuges eingegangen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Zielsetzung

Bei der Programmausführung wird in den meisten Fällen ein gewisses Ergebnis verfolgt, der Programmablauf kann aber ausnahmsweise noch vom grösseren Interesse sein. Bei nebenläufigen Programmen mit nicht-deterministischer Reihenfolge ist es von grosser Bedeutung, Informationen während des Programmablaufs darzustellen. Beispiele von Informationen sind die Anzeige von Zustandsänderungen oder Nachrichtenaustausch.

Die Visualisierung von Programmen, die animierte Darstellung unterschiedlicher Aspekte der Programmausführung, wurde eingesetzt, um die Komplexität des Ablaufs bewältigen zu können. Insbesondere wurde die Programmvisualisierung auf Gebiete angewendet wie Fehlersuche, Lernprogramme, Überprüfung oder Verständnis von Programmen.

Das JPict Framework ist eine nicht-verteilte Implementierung der Operationen des asynchronen π-Kalkülus [1]. Mit Hilfe von JPict wurde die Applikation PiL entwickelt. Die mit PiL geschriebenen Programme werden in der vorliegenden Arbeit π-Programme genannt. Der Einsatz des in dieser Arbeit entwickelten Visualisierungswerkzeuges soll die Frage nach den Informationen über den Programmablauf eines π-Programmes beantworten. Folgende Hauptziele lassen sich deshalb für das Visualisierungswerkzeug formulieren:

1. Es soll ein interaktives Werkzeug darstellen, das die Ausführung eines π-Programmes schrittweise ermöglicht.
2. Die wichtigsten Programmzustände sollen hervorgehoben werden, so dass der Endbenutzer die Zustandsänderungen gleichzeitig mitbekommt.
3. Der Austausch von Nachrichten soll angezeigt werden.
4. Es soll die Dynamik (z.B. Erzeugen oder Löschen von Objekten) eines π-Programmes unterstützen.

Durch die Realisierung dieses Visualisierungswerkzeuges wird möglich, den Ablauf eines π-Programmes besser zu verstehen, die Dynamik zu verfolgen und Fehler zu beheben.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit ist in folgende Kapitel unterteilt:

Das JPict Framework
Das JPict Framework [2] ist eine nicht-verteilte Implementation des asynchronen π-Kalkülus [1]. Die Implementation verwendet die Technik mehrerer parallelen Kontrollflüsse (engl. *threads*), die von Java unterstützt wird.

Das JHotDraw Framework
HotDraw [3] ist ein Framework für objektbasierte grafische zweidimensionale Editoren, das von Kent Beck und Ward Cunningham in Smalltalk entwickelt wurde. Als Framework, stellt es nicht nur einfachen Code dar, sondern, durch Einsetzten von Patterns, auch ein flexibes, wartbares und wiederverwendbares Design [4].

Dieses Projekt verwendet die Implementation von Erich Gamma und Thomas Eggenschwiler, **JHotDraw** [5] genannt, die ursprünglich als eine Fallstudie für der Einsatz von Design Patterns entwickelt worden ist. Der Vorteil der Implementierung in Java ist, dass das Framework-Design nur durch *interfaces* repräsentiert wird, was eine Trennung zwischen Konzept und Implementierung ermöglicht.

Visualisierungsmodelle
In diesem Kapitel wird auf verschiedene Visualisierungsmodelle eingegangen und das Modell, worauf diese Arbeit beruht, dargestellt.

Design von VisualPi
In diesem Kapitel wird das Design der vorliegenden Arbeit modulweise dargestellt.

Implementation von VisualPi
Hier wird auf die Implementation zur Hervorhebung wichtiger Aspekte eingegangen.

Anwendung von VisualPi
In diesem Kapitel wird die Anwendungsweise von VisualPi anhand von drei Beispielen dargestellt.

Schlussfolgerungen
Das letzte Kapitel schliesst diese Arbeit mit einer Zusammenfassung. Erkenntnisse und meine Erfahrungen während der Entwicklung werden beschrieben. Der Ausblick am Schluss zeigt Erweiterungsmöglichkeiten dieser Arbeit auf.

1.3 Bedeutung des Visualisierungswerkzeuges

VisualPi, das Visualisierungswerkzeug, das während dieser Arbeit in Java entwickelt wurde, verwendet das JHotDraw Framework, um den Ablauf eines π-Programmes darzustellen. Dieses Werkzeug wurde als separate Schnittstelle zwischen den beiden Frameworks eingebaut (Abbildung 1.1) und verwendet den grafischen Rahmen, den JHotdraw zur Verfügung stellt um, die Semantik eines π-Programmes mittels neu definierten grafischen Objekten hervorheben zu können.

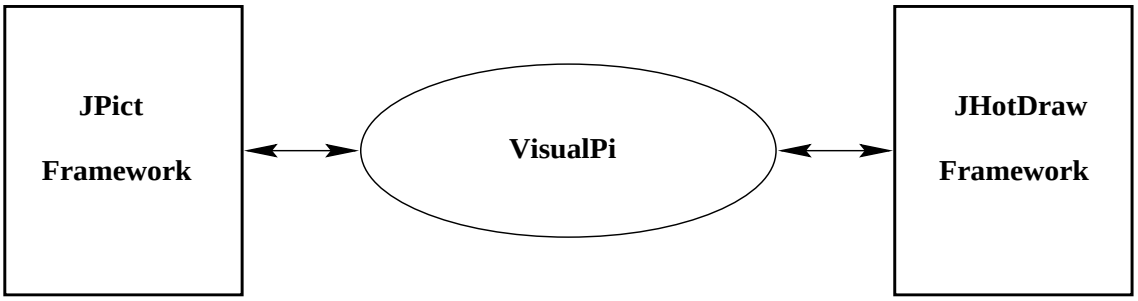

Abbildung 1.1: Stellung des Visualisierungswerkzeuges - VisualPi

1.4 Eingesetzte Hilfsmittel

Die Entwicklung des benötigten Codes für dieses Projekt wurde auf der Hardwareplattform Sun Sparc (SunOS) durchgeführt. Zum Einsatz kamen dabei die Entwicklungsumgebung SNIFF+, in welcher u.a. ein Java-Compiler integriert ist. Bei der Erstellung der Klassendiagramme wurde die letzte Version der UML-Notation [6] verwendet.

1.4.1 Verwendete Schrifttypen

Ausser der normalen Schrift werden in der vorliegenden Dokumentation noch zwei andere Schrifttypen verwendet:

- Kursive Schrift: Wird verwendet um Konzepte oder unübersetzte englische Begriffe, die einem höheren Abstraktionsniveau entsprechen, vom gewöhnlichen Text zu unterscheiden. Beispiele dafür sind: *interface*, das ein reserviertes Wort in Java ist, oder *thread*.
- Schreibmaschinenschrift: Wird für ein tieferes Niveau, die Implementationsebene, eingesetzt, wenn es sich um Klassen-, Methodennamen oder Programmcode handelt. Ein Beispiel dafür ist `DefaultChannel`, ein Klassenname im JPict Framework.

Kapitel 2

Das JPict Framework

Das JPict Framework ist eine Implementation in Java des asynchronen π-Kalkülus [1]. Der grösste Teil des Frameworks beschäftigt sich mit nicht-verteilten Agenten (engl. *agents*), die Werte über Kanäle (engl. *channels*) austauschen. In diesem Kapitel wird kurz auf die wichtigsten Konzepte eingegangen, die unentbehrlich für die vorliegende Arbeit sind. Eine ausführliche Beschreibung ist der Arbeit [2] zu entnehmen.

2.1 Einführung

PiL ist eine Applikation, die mit Hilfe von JPict Framework entwickelt wurde. Die Schnittstelle zum Benutzer enthält eine Eingabesprache [7], in der π-Programme geschrieben werden. Das π-Programm wird in einem Fenster (das weisse Rechteck in der Abbildung 2.1) von Benutzer eingegeben, das von PiL zur Verfügung gestellt wird.

Das π-Programm kann nur in einem Schritt ausführt werden, in dem der Benutzer auf den ” execute ”-Knopf drückt. In diesem Moment wird der PiL-Interpreter gestartet, der das Programm interpretiert und die entsprechenden *threads* startet. Die Programmausgabe folgt in einem zweiten Fenster, dem unteren Rechteck in der Abbildung 2.1. Die weiteren Knöpfe im Eingabefenster (Abbildung 2.1) bewirken:

- ”clear” - Das Fenster, in dem die Ausgabe des Programmes folgt, wird geleert.
- ”load” - Ein existierendes π-Programm (Endung ” pil ”) wird geöffnet.
- ”save” - Ein geöffnetes π-Programm wird gespeichert.
- ”save as” - Ein geöffnetes π-Programm wird unter einem anderen Namen gespeichert.
- ”new” - Ein neues π-Programm wird erzeugt.

Im Beispiel, das in der Abbildung 2.1 vorgestellt wird, wird in der ersten Zeile des π-Programmes durch die ”extern”-Deklaration ein externer Kanal geladen, um die Ausgabe von Ergebnissen im Ausgabefenster zu ermöglichen. Der Kanal heisst ”pr”. In den nächsten zwei Zeilen werden durch Verwendung des

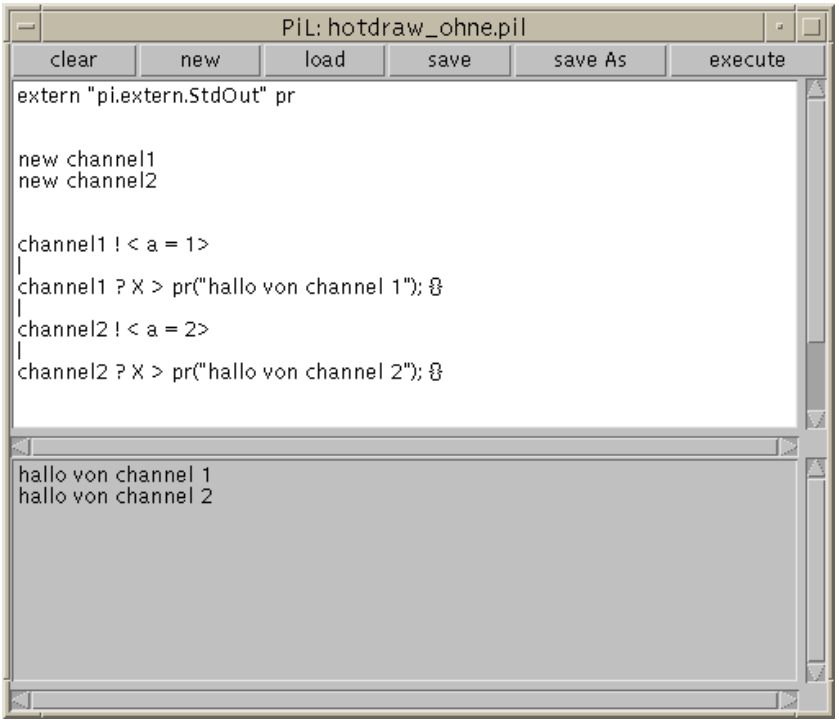

Abbildung 2.1: JPict Schnittstelle zum Benutzer

reservierten Wortes ”new” zwei Kanäle, ”channel1” und ”channel2”, erzeugt. Die nächsten Zeilen stellen mehrere *Parallel Agents* [2] dar. *Parallel Agent* bedeutet, dass dieser Agent andere Agenten enthält, die parallel laufen. In diesem Beispiel sind vier Agenten enthalten:

- `channel1 ! <a = 1>`

  ein *Send Agent*, der eine *put*-Operation (das Ausrufezeichen) auf den ”channel1” ausführt, indem die Form ”<a = 1>” in die Wertwarteschlange des ”channel1” gelegt wird.

- `channel1 ? X > pr("hallo von channel 1"); {}`

  ein *Receiving Agent*, der eine *get*-Operation (das Fragezeichen) auf den ”channel1” ausführt, indem die Form ”<a = 1>” aus der Wertwarteschlange des ”channel1” geholt wird und mit dem Bezeichner ”X” verbunden wird. Das Zeichen ”>” deutet darauf hin, was anschliessend ausgeführt wird. In diesem Fall wird über den Kanal ”pr” die Ausgabe, ”hallo von channel 1”, durchgeführt.

- `channel2 ! <a = 2>`

  ein *Send Agent*, der eine *put*-Operation auf den ”channel2” ausführt.

- `channel2 ? X > pr("hallo von channel 2"); {}`

  ein *Receiving Agent*, der eine *get*-Operation auf den ”channel2” ausführt.

Die Nebenläufigkeit der Agenten ist durch das Zeichen ”|” angegeben. Die Syntax des PiL-Interpreters ist ausführlich in [7] beschrieben worden.

2.2 Agenten und Kanäle

Agenten [2] sind der aktive Teil des Frameworks. Sie können als Miniprozesse aufgefasst werden, die eine bestimmte Aktion durchführen. Je nach Aktion, haben sie eine andere Bezeichnung. Beispiele dafür sind:

- *Send Agent* - Senden von Werten über Kanäle.
- *Receiving Agent* - Empfangen (Lesen) von Werten von einem Kanal.
- *Parallel Agent* - Führt parallel andere Agenten.

Die Agenten werden in *threads* ausgeführt und, solange sie aktiv sind, enthalten sie eine Umgebung. Die Aufgabe der Umgebung ist Bezeichner auf Werte abzubilden.

Die Agenten kommunizieren mittels Kanälen miteinander. Ein Kanal enthält eine Wertwarteschlange und eine Agent-Warteschlange. Ein *Send Agent* kann Werte in die Warteschlange mit der *put*-Operation legen und ein *Receiving Agent* kann diese Werte mit der *get*-Operation holen. Der Wert bleibt in der Warteschlange, solange ihn kein Agent benötigt. Ein Kanal kann sich in einem von den 3 Zuständen befinden (Abbildung 2.2):

- **nonempty**: die Wertwarteschlange des Kanals enthält Werte. Die *put*- und *get*-Operationen bewirken:
  - neues *put*: Zustand bleibt unverändert, denn es wird nur ein neuer Wert der Warteschlange hinzugefügt.
  - neues *get*: einer von den zwei Fällen kann auftreten:
    - Übergang nach Zustand *empty*, weil der Wert, der geholt wird, der letzte Wert in der Warteschlange ist,
    - Zustand bleibt unverändert, weil noch weitere Werte in der Warteschlange sind.
- **empty**: die Wertwarteschlange ist leer. Die zwei Operationen verursachen:
  - neues *put*: Übergang nach Zustand *nonempty*, weil die Wertwarteschlange nicht mehr leer ist,
  - neues *get*: Übergang nach Zustand *blocking*, weil kein Wert in der Warteschlange ist. Null wird ausgegeben und der Agent muss auf einen Wert warten.
- **blocking**: die Wertwarteschlange ist leer und die Agent-Warteschlange ist nicht leer. Hier bewirken die zwei Operationen folgendes:
  - neues *put*: Übergang nach Zustand *nonempty*, weil ein Wert in die Wertwarteschlange gelegt wird,
  - neues *get*: Zustand bleibt unverändert, weil der Agent, der die Operation durchgeführt hat, der Agent-Warteschlange hinzugefügt wird.

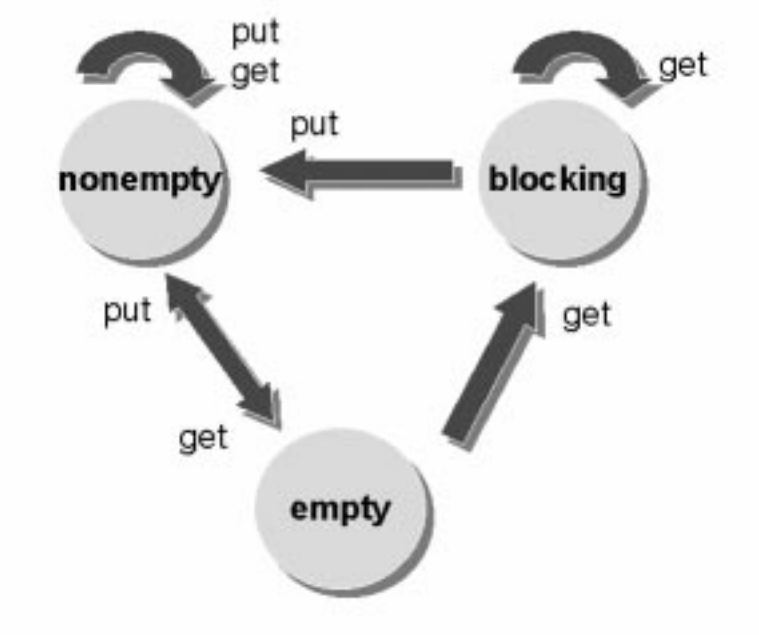

Abbildung 2.2: Kanalzustände und -zustandsänderungen

2.3 Formen

Die Werte, die in PiL zwischen den Agenten ausgetauscht werden, haben ein spezielles Format, names Form (engl. *form*) [2]. Eine Form ist eine endliche Abbildung zwischen einer Bezeichnung (engl. *label*) und einem Kanal, Zahl oder String. Eine Form sieht wie folgt aus:

$$< label_1 = value_1, label_2 = value_2, ...label_n = value_n >$$

Kapitel 3

Das JHotDraw Framework

3.1 Einführung

Mit Hilfe von JHotDraw Framework [5] kann man objektbasierte grafische Editoren aufbauen, die eine zweidimensionale Darstellung beliebiger Diagramme, Entwürfe usw. unterstützen. Neben der interaktiven Erstellung der Zeichnung durch spezielle Werkzeuge besteht die Möglichkeit, Objekte aus einer existierenden Umgebung zu visualisieren. Die Umgebung ist in unserem Fall das JPict Framework.

Eine JHotDraw-Anwendung bearbeitet Zeichnungen, die aus verschiedenen elementaren Zeichnungselementen bestehen. Zeichnungselemente sind zum Beispiel Linien, Ellipsen, Rechtecke, Text, Bezier-Kurven und vieles mehr. Zwei wichtige *interfaces* des JHotDraw Frameworks sind daher `Drawing` (die Zeichnung) und `Figure` (die Objekte, die in der Zeichnung enthalten sind).

Ein JHotDraw-Editor (Abbildung 3.1) beinhaltet eine Menge von Werkzeugen. Werkzeuge werden eingesetzt um die Zeichnung interaktiv zu manipulieren. Sie werden links neben der Zeichnung in der Palette durch jeweils ein Symbol dargestellt und durch das *interface* `Tool` beschrieben.

Wird ein Zeichnungselement mit Hilfe des Selektionswerkzeuges ausgewählt, so wird eine dem Zeichnungselement zugehörige Menge von *handles* dargestellt. *Handles* stellen eine Möglichkeit dar, die Eigenschaften eines Zeichnungselements mit der Maus zu verändern oder spezifische Aktionen durchzuführen. *Handels* sind in der Abbildung 3.1 in Form kleiner Rechtecke rund um das grosse Rechteck zu sehen. Die Beschreibung von Handles findet man im *interface* `Handle`.

Zeichnungselemente können auf verschiedenste Art und Weise miteinander verknüpft werden. Derartige Verknüpfungen werden als *constraints* bezeichnet. Mit *constraints* können beispielsweise Zeichnungselemente einfacher positioniert oder verbunden werden oder es kann gewährleistet werden, dass zwei Zeichnungselemente immer die gleiche Grösse oder Farbe besitzen. Diese Eigenschaft wird durch den Einbau einer Rasterung, die auf das *interface* `PointConstrains` basiert, zur Verfügung gestellt.

Das *interface* `DrawingEditor` repräsentiert den Standard-Editor von JHotDraw und dient als

Grundlage für alle weiteren Editoren. `DrawingEditor` kann als eigenständige Anwendung fungieren oder Teil einer grösseren Java-Anwendung sein.

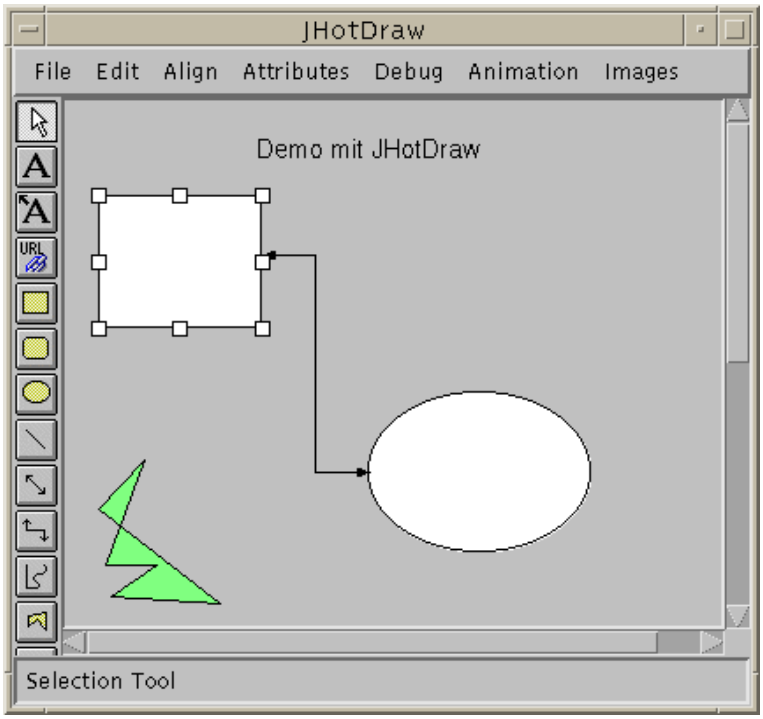

Abbildung 3.1: Ein JHotDraw Editor

3.2 Design Beschreibung

Die Architektur des JHotDraw Frameworks kann man in 3 logische Ebenen teilen:

- **Design Ebene**
  Diese Ebene ist durch folgende *interfaces* implementiert:
  - `DrawingEditor`
  - `DrawingView`
  - `Drawing`
  - `Figure`
  - `Tool`
  - `Handle`

  Ein *interface*-Diagramm mit den entsprechenden Beziehungen wird in der Abbildung 3.2 dargestellt.

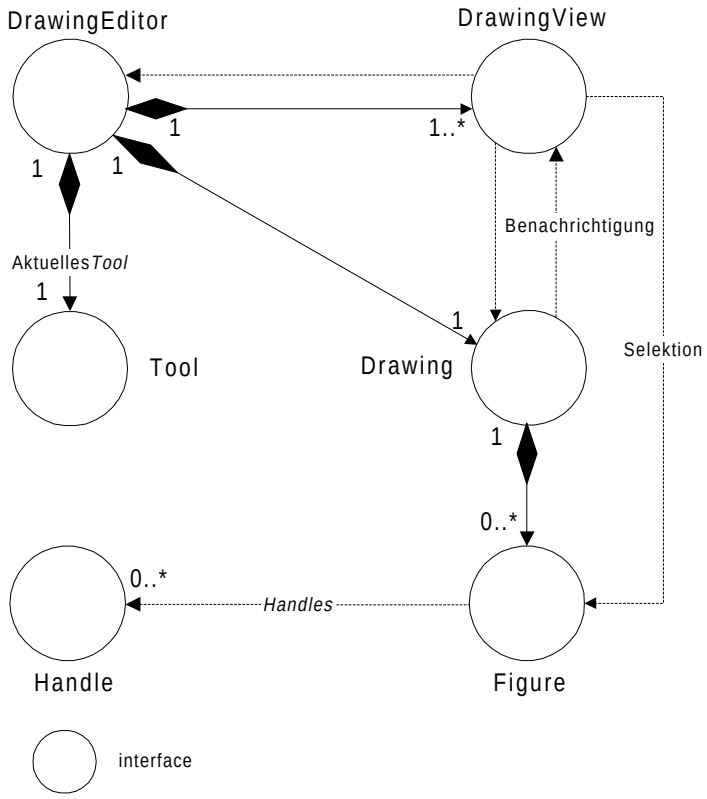

Abbildung 3.2: Design Ebene: *interface*-Diagramm

- **Standard Ebene**
  Hier (Abbildung 3.3) sind zum Teil abstrakte Klassen enthalten, die die obigen *interfaces* verwenden:
  - `DrawApplication` oder `DrawApplet`
  - `StandardDrawingView`
  - `StandardDrawing`
  - `AbstractFigure`
  - `AbstractTool`
  - `AbstractHandle`

- **Die eigentliche Implementation**
  Auf dieser Ebene ist der ganze Code enthalten, der zum grössten Teil von der Standard Ebene

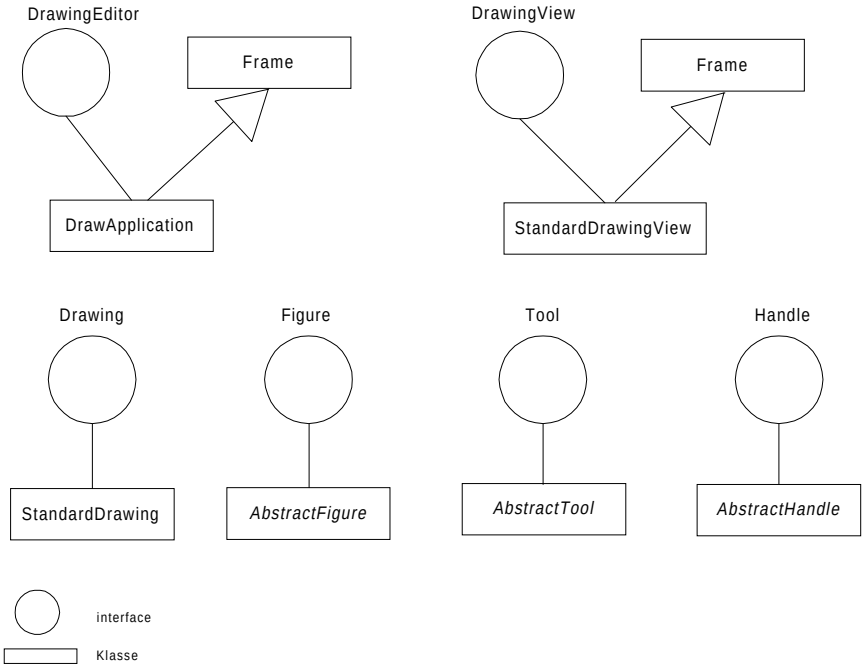

Abbildung 3.3: Design- und Standard-Ebene

abgeleitet und erweitert wurde.

3.2.1 DrawingEditor

`DrawingEditor` (Abbildung 3.2) definiert die zentrale Stelle für die Koordination unterschiedlicher Komponenten, die in einem Zeichnung-Editor teilnehmen. Im *Mediator* Pattern spielt `DrawingEditor` die *Mediator* Rolle. Er entkoppelt die Teilnehmer eines Zeichnung-Editors.

3.2.2 DrawingView

`DrawingView` (Abbildung 3.2) ist dafür verantwortlich, die Zeichnung in einem Fenster auf dem Bildschirm darzustellen. `DrawingView` spielt im

- *Observer* Pattern die Rolle des Beobachters (*Observer*). `DrawingView` muss konsistent mit `Drawing` sein, und deshalb verfolgt es die Änderungen der Zeichnung (`Drawing`), die die Rolle des Subjektes (*Subject*) spielt.
- *State* Pattern die Rolle von *StateContext*. `Tool` hat die *State* Rolle.
- *Strategy* Pattern die Rolle von *StrategyContext* in bezug auf die *UpdateStrategy*. *UpdateStrategy* ist dafür verantwortlich die Darstellung (`DrawingView`) flimmerfrei anzuzeigen, wenn die Zeichnung (`Drawing`) aktualisiert wird. Dieses Ziel wird durch eine doppelte Pufferung der Zeichnung

erreicht. Die Zeichnung wird zunächst in einer Pixmap (rechteckiger, unsichtbarer Pixel-Bereich im Speicher) aufgebaut und anschliessend auf den Bildschirm kopiert.

3.2.3 Drawing

`Drawing` (Abbildung 3.2) enthält die Zeichnungselemente (`Figures`). Auf die Zeichnungselemente kann zugegriffen werden mittels Operationen (z.B. `add, remove, findFigure`), die `Drawing` zur Verfügung stellt. `Drawing` spielt im *Observer* Pattern die Rolle des Subjektes (*Subject*). Durch Einsetzen dieses Patterns wird erreicht, dass `Drawing` von den Darstellungen (`DrawingViews`) entkoppelt wird und somit mehrere Darstellungen für ein `Drawing` gleichzeitig existieren können. Wenn ein Teil von `Drawing` ungültig wird, benachrichtigt `Drawing` alle `DrawingViews`.

3.2.4 Figure

`Figure` (Abbildung 3.2) repräsentiert die Grundlage für alle grafischen Objeckte, die in einer JHotDraw Applikation auftreten können. `Figure` kennt seine Darstellungsbox (engl. *display box*). Die Darstellungsbox ist ein Rechteck, das die Position eines `Figure` auf dem Bildschirm angibt. Der Aufbau eines `Figure` kann verschachtelt sein, indem es andere Zeichnungsobjekte enthalten kann. Die Schnittstelle eines `Figure` enthält `Handles` und `Connectors`, die ermöglichen, dass man es von aussen manipulieren kann. Mittles `Handles` kann man die Form oder die Attribute eines `Figure` definieren und ändern. `Connectors` sind für die Definition und Lokalisierung von Verbindungspunkten (engl. *connection points*) eines `Figure` zuständig.

3.2.5 Tool

`Tool` (Abbildung 3.2) definiert einen Modus von `DrawingView`. Das bedeutet, dass alle Eingabeereignisse, die an `DrawingView` gerichtet werden, an das aktive `Tool` weitergeleitet werden. Somit definiert das selektierte (aktive) `Tool`, wie die Eingabe behandelt (engl. *parsed*) wird. Wenn `Tool` seine Aufgabe beendet hat, informiert er den `DrawingEditor` durch Aufruf der `DrawingEditor`-Methode "`toolDone()`". Die `Tools` werden gleichzeitg mit dem `DrawingEditor` erzeugt, werden immer wieder verwendet, aktiviert und deaktiviert. Im *State* Pattern spielt `Tools` die *State* Rolle und `DrawingView` diejenige von *StateContext*.

3.2.6 Handle

`Handles` (Abbildung 3.2) werden benötigt, um ein Zeichnungsobjekt (`Figure`) durch direkte Manipulation zu verändern. `Handles` kennen `Figure`, zu dem sie gehören, und stellen Methoden zur Verfügung, um ein `Handle` auf einem `Figure` zu lokalisieren und Änderungen des `Figure` zu verfolgen. Im *Adapter* Pattern spielt `Handle` die *Adapter* Rolle und stellt damit eine einheitliche Schnittstelle zwischen alle Typen von Zeichnungsobjekte und `DrawingView`. Dadurch wird gewährleistet, dass Benutzerereignisse, die ein `Figure` betreffen, wie z.B. Verschieben oder Vergrössern eines `Figure`, in Änderungen des entsprechenden Zeichnungsobjekt umgewandelt werden.

3.3 Bedeutung des JHotDraw Frameworks

Das JHotDraw Framework stellt aus folgenden Gründen einen gut definierten Rahmen für die vorliegende Arbeit dar:

- Das Design von JHotDraw Framework wurde nur auf Pattern Basis entwickelt, was die Verständlichkeit dieses Frameworks erleichtert.
- Weil JHotDraw in Java implementiert wurde, bedeutet es, dass Java die gemeinsame Plattform für alle drei Komponenten (JPict, JHotDraw und VisualPi) ist. Diese Tatsache erleichtert die Kommunikation zwischen den Komponenten und ermöglicht eine einfache Integration.
- JHotDraw stellt bereits Komponenten zur Verfügung, die für diese Arbeit benötigt werden. Auf der einen Seite wird ein Teil der Komponenten, wie z.B. `DrawingView` und `Drawing`, unmodifiziert in der vorliegenden Arbeit verwendet, auf der anderen Seite werden Komponenten, wie z.B. `DrawingEditor, Figure, Tool`, für die Bedürfnisse dieser Arbeit erweitert und spezialisiert.

Kapitel 4

Visualisierungsmodelle

Programmvisualisierungen ermöglichen ein besseres Verständnis paralleler Programme. Bei parallelen Programmen entspricht das semantische Verhalten eines Algorithmus verschiedenen Reihenfolgen von *low-level* Ereignissen. Visualisierungen liefern insbesondere Abstraktionsmechanismen für das *high-level* Verhalten, währenddem andere typische Analysierungswerkzeuge dazu neigen ausführliche *low-level* Abläufe darzustellen.

4.1 Das *State-Transition* Modell

Das *State-Transition* Modell [8] ist das klassische Visualisierungsmodell, bei dem die Ablaufzustände auf eine grafische Darstellung abgebildet werden. Bei dem Aufbau einer solchen Abbildung werden drei Schritte durchgeführt:

1. Identifizierung von aussagekräftigen Programmzuständen.
2. Festlegung von Visualisierungsobjekten, die diesen Programmzuständen entsprechen.
3. Definition eines Abbildungsmechanismus zwischen den Visualisierungsobjekten und Programmzuständen.

In diesem Modell wird die Visualisierung mit dem Zustandsübergang des Programmes synchronisiert. Wenn dieses Modell für parallele Programme angewendet wird, braucht es eine globale Aufnahme des ganzen Zustandes. Eine globale Zustandsaufnahme ist im Fall von parallelen Programmen unmöglich zu realisieren, weil parallele Programme nicht-deterministisch sind, d.h., sowohl das Ergebnis als auch die Zustansänderungen während des Programmablaufs sind nicht eindeutig.

4.2 Das *Actor Event* Modell

4.2.1 Modellbeschreibung

Actors [9] bedeuten in diesem Modell verkapselte, interaktive und selbständige Komponenten, die mittels asynchronen Nachrichten miteinander kommunizieren. Ein *Actor* ist definiert durch einen Namen und

eine Nachrichtenwarteschlange, in der die empfangenen Nachrichten aufbewahrt werden.
Ein *Actor Event* entsteht, wenn ein *Actor* eine der folgenden Aktionen durchführt:

1. Nachrichten zu anderen *Actors* schicken oder von anderen *Actors* empfangen,
2. neue *Actors* erzeugen,
3. eigenen Zustand ändern.

Das *Actor Event* Modell [8] bildet Visualisierungsereignisse auf Visualisierungsaktionen durch Visualisierungsmechanismen ab. Dei drei neu eingeführten Begriffe bedeuten:

Visualisierungsereignis
Definiert die *Actor Events*, d.h., die Ereignisse, die im System auftreten, die zu visualisieren sind.

Visualisierungsaktion
Entspricht einer Darstellungsaktion oder Animation auf dem Visualisierungsmonitor, der dafür verantwortlich ist die Anzeige auf dem Bildschirm zu generieren.

Visualisierungsmechanismus
Ist dafür verantwortlich die entsprechenden Visualisierungsaktionen beim Auftreten eines Visualisierungsereignises auszulösen.

Dieses Modell (Abbildung 4.1), das zur Visualisierung der *Actor Events* dient, stellt ein Verfahren zur Verfügung, um die *Actor Events* zu detektieren und die entsprechenden Visualisierungsaktionen auszulösen. Das Modell hat folgende Merkmale:

- Der Visualisierungsmechanismus ist verteilt, so dass jeder *Actor* von einem separaten *Observer* beobachtet wird (Abbildung 4.1). Das heisst, dass ein *Observer* die *Actor Events* lokal entdeckt.
- Weil die *Actor Events* lokal überwacht werden, entfällt die Notwendigkeit von globalen Aufnahmen des ganzen Programmzustandes, wie beim *State-Transition* Modell.

4.2.2 Modellerweiterung

Das *Actor Event* Modell ist einfach, elegant und kann für parallele Programme gut eingesetzt werden, die nicht sehr komplex sind. Steigt die Komplexität dieser Programme, so wird dieses Modell wegen zu vielen *Observers* unübersichtlich, die die Aktivität jedes *Actor* beobachten. In solchen Situationen ist das Modell*Causal Interaction* zu verwenden. Es basiert auf demselben Prinzip, das im vorherigen Abschnitt dargestellt wurde, mit folgenden Erweiterungen:

- Die *Actors* werden in Visualisierungsgruppen zusammengefügt.
- Neben den *Observers* werden auf eine höhere Ebene *Coordinators* eingefügt, die eine Visualisierungsgruppe beobachten.
- Die Visualisierungsereignisse werden in Patterns strukturiert und als solche von den *Coordinators* detektiert.

Diese Erweiterungen führen eine Art Zentralisierung ein, die das Ziel hat, die Übersicht durch Gruppierung in komplexen parallelen Systemen zu behalten. Hier wird weiter auf dieses Modell nicht eingegangen. Mehr dazu ist in [8] zu finden.

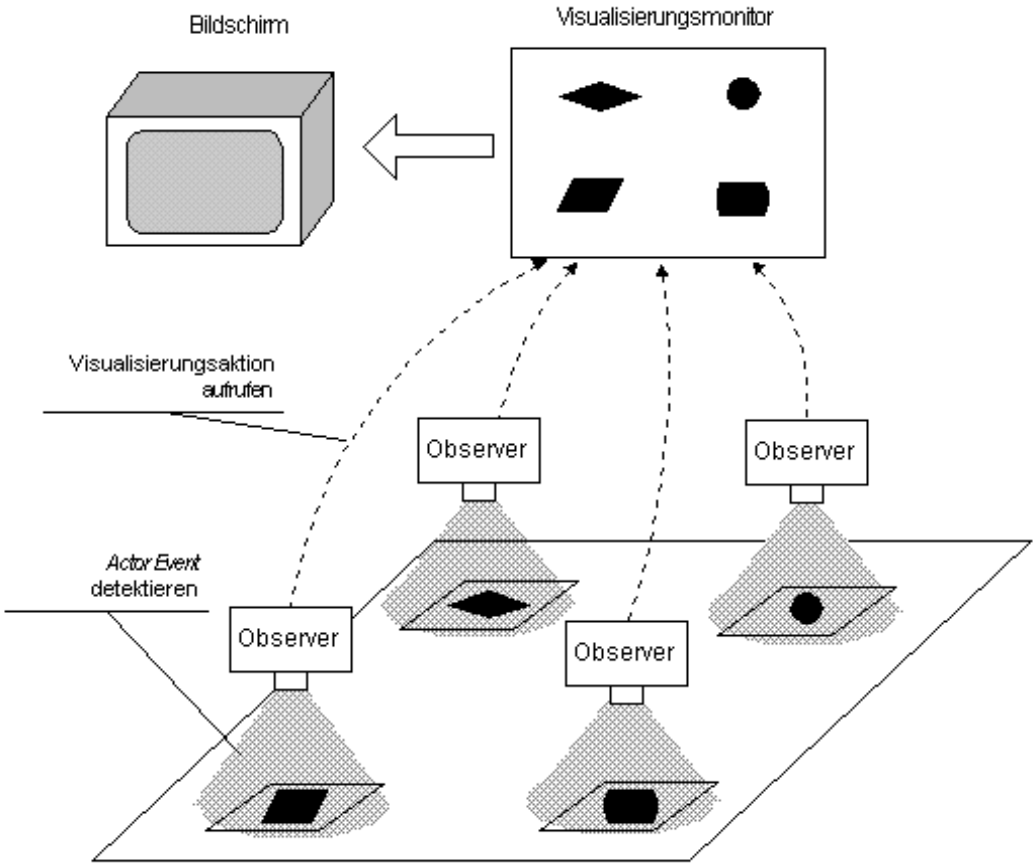

Abbildung 4.1: *Actor Event* Visualisierungsmodell

4.2.3 Bedeutung des *Actor Event* Modells

Das *Actor Event* Modell wurde in der vorliegenden Arbeit verwendet, um die Dynamik des asynchronen π-Kalkülus zu wiederspiegeln. Die Kanäle aus dem JPict Framework werden auf die *Actors* abgebildet. Durch das Einsetzen dieses Modells wurden folgende Abbildungen für die im Abschnitt 4.2.1 dargestellten drei Aktionstypen definiert:

- die Agent-Operationen *put* und *get*: entsprechen der Aktion 1 (Nachrichten schicken und empfangen) .
- Kreieren neuer Kanäle: entspricht der Aktion 2 (neue *Actors* erzeugen).
- Zustandsänderungen der Kanäle: entsprechen der Aktion 3 (eigenen Zustand ändern).

Kapitel 5

Design von VisualPi

Dieses Kapitel stellt das Design des Visualisierungswerkzeuges VisualPi vor. Zuerst werden die Anforderungen an das Werkzeug definiert, danach wird schrittweise auf das Design eingegangen.

5.1 Anforderungen

Das Hauptziel des VisualPi Werkzeuges ist, den dynamischen Ablauf eines π-Programmes zu visualisieren. Ein π-Programm ist ein Programm, das mit Hilfe der PiL-Applikation geschrieben wurde. Der dynamische Ablauf kann durch folgende Grundzüge beschrieben werden:

- Agenten und Kanäle werden während der Ausführung erzeugt.
- Formen werden zwischen den Agenten über Kanäle asynchron ausgetauscht, durch Ausführung der *put*- und *get*-Operationen.
- Durch Ausführung der *put*- und/oder *get*-Operationen ändern die Kanäle ihren Zustand, der *non-empty, empty* oder *blocking* sein kann.

Der Zustand bei der Ausführung eines π-Programmes vor dem Einsetzen des VisualPi Werkzeuges war folgender:

- die einzige Möglichkeit war, das ganze Programm in einem Schritt ausführen.
- die Ausführung lieferte nur ein Endergebnis, eventuell in Form einer Textausgabe.
- die Dynamik während des Programmablaufes blieb versteckt.

Folgende Anforderungen lassen sich für VisualPi definieren:

- der Kanal ist das Hauptelement, das visualisiert wird. Die Gründe zu diesem Entscheid sind, dass auf der einen Seite die Kommunikation zwischen den Agenten nur über Kanäle stattfindet, auf der anderen Seite, dass sie vom Kanalzustand abhängig ist (Abschnitt 2.2).
- Das π-Programm soll schrittweise ausgeführt werden, so dass jedes Ergebnis einer *put*- oder *get*-Operation angezeigt wird.

- Der wichtigste Kanalzustand *nonempty*, in dem die Wertwarteschlange Formen enthält und Agenten darauf warten *get*-Operationen auszuführen, soll hervorgehoben werden.
- Bei der Erzeugung der Darstellungsobjekte soll ihre Positionierung im Anzeigefenster auf dem Bildschirm vom π-Programm aus angegeben werden.
- Es soll möglich sein die Darstellungsobjekte im Anzeigefenster vom π-Programm aus oder interaktiv mit der Maus zu löschen oder zu verschieben.
- Nachrichten, die zwischen den Kanälen ausgetauscht werden, sollen dargestellt werden.

5.2 Design-Module

Das Design des VisualPi Werkzeuges kann in folgende Module eingeteilt werden:

- der Kern: enthält die Definition der Schnittstellen zu den JPict und JHotDraw Frameworks.
- Erzeugung und Positionierung der Visualisierungsobjekte.
- Anzeige der Kanalzustände.
- Interaktive Ausführung eines π-Programmes.
- Anzeige der Kanal-Wertwarteschlange.
- Dienste.
- Anzeige des Nachrichtenaustausches zwischen den Kanälen.

5.3 Der Kern von VisualPi

Der Kern ist aus den Schnittstellen zu den JPict und JHotDraw Frameworks zusammengesetzt. Beim Design wurde berücksichtigt, dass die Schnittstellen durch so wenige Klassen wie möglich definiert sein sollen, damit man die Übersicht und den Aufwand nachträglicher Änderungen der beiden Frameworks im VisualPi so klein wie möglich behalten kann.

5.3.1 Die Schnittstelle zu JPict

Die Klassen, die hier definiert wurden, benutzen nur drei der abstrakten Klassen, bzw. *interfaces*, die JPict zur Verfügung stellt:

- `Function` ist eine abstrakte Klasse, mit deren Hilfe Funktionen als externe Elemente, bezogen auf PiL, erzeugt werden können. ”Extern” bedeutet, dass sie explizit in einem π-Programm aufgerufen werden können. `Function` enthält die abstrakte Methode `invoke()`, die von den Unterklassen überschrieben wird. `invoke()` enthält den Code, der beim Aufruf der ensprechenden Funktion ausgeführt wird.

- `Channel` ist ein *interface* und definiert die zwei Methoden `put()` und `get()`, die den im Abschnitt 2.2 vorgestellten zwei Operationen *put* und *get* entsprechen.
- `DefaultChannel`, eine Klasse, die `Channel` implementiert, ist dafür verantwortlich, die Werte, die zwischen den Agenten ausgetauscht werden, in der Wertwarteschlange und wartende Agenten in der Agent-Warteschlange zu behalten. `DefaultChannel` ist die Umsetzung auf die tiefere Ebene des Konzeptes "Kanal", der im Abschnitt 2.2 eingeführt worden ist.

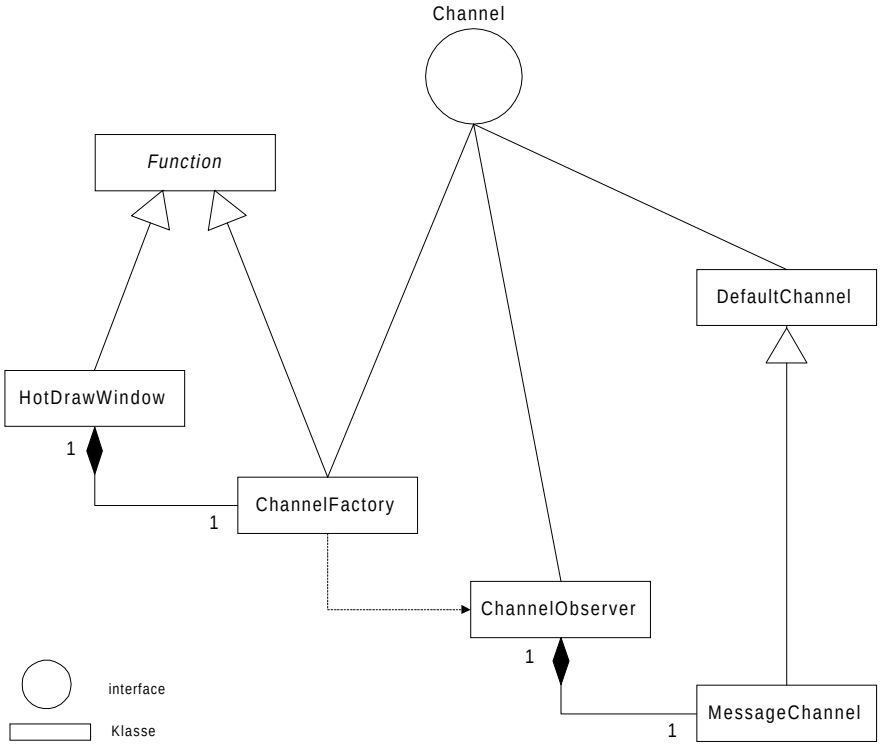

Abbildung 5.1: Die Schnittstelle zu JPict

Die Schnittstelle zu JPict enthält folgende Klassen, die auch im Klassendiagramm 5.1 samt Beziehungen dargestellt werden:

- `HotDrawWindow`: Stellt die höchste Ebene dieser Schnittstelle dar, und ist dafür verantwortlich, den VisualPi-Editor (GUI) zu erzeugen. Dafür wird ein Objekt der Klasse `ChannelFactory` kreiert. `HotDrawWindow` ist eine Unterklasse der abstrakten Klasse `Function` und implementiert die `invoke()`-Methode.
- `ChannelFactory`: Ist für die Erzeugung des VisualPi-Editors `ChannelDrawApplication` und der grafischen Objekte (`ChannelObservers`), die den Kanälen entsprechen, verantwortlich. Dabei werden die Objekte im VisualPi-Editor dargestellt. `ChannelFactory` ist ebenfalls eine Unterklasse der abstrakten Klasse `Function` und definiert die `invoke()`-Methode. Beim Aufruf der `invoke()`-Methode wird ein `ChannelObserver`-Objekt kreiert.

- `ChannelObserver`: Ist das Darstellungsobjekt in VisualPi für einen Kanal. Im *Actors Event* Modell, das in der vorliegenden Arbeit eingesetzt wurde, spielt `ChannelObserver` die Rolle des *Observers*, während der Kanal die *Actor*-Rolle spielt (Abschnitt 4.2.3). An dieser Stelle sei folgendes bemerkt: der Name der Klasse `ChannelObserver` bezieht sich auf die *Observer*-Rolle im *Actors Event* Modell und nicht auf diejenige im *Observer* Pattern. Die Beziehung zwischen `ChannelObserver` und dem Kanal wird im Unterkapitel 5.4.1 vorgestellt.

- `MessageChannel`: Hat dieselbe Rolle wie `DefaultChannel`, nämlich Werte, die zwischen den Agenten ausgetauscht werden, in der Wertwarteschlange und wartende Agenten in der Agent-Warteschlange zu behalten. Der Grund, weshalb `MessageChannel` von `DefaultChannel` abgeleitet wurde, ist die Notwendigkeit Funktionalitätserweiterungen für VisualPi einzuführen. Beispiele von Erweiterungen sind Zugrifferlaubnisse für die zwei Warteschlagen der Klasse `DefaultChannel`.

Ein Zeitdiagramm mit der Reihenfolge, in der Objekte der oben beschriebenen Klassen während der Ausführung eines π-Programmes erzeugt werden, wird in der Abbildung 5.2 dargestellt.

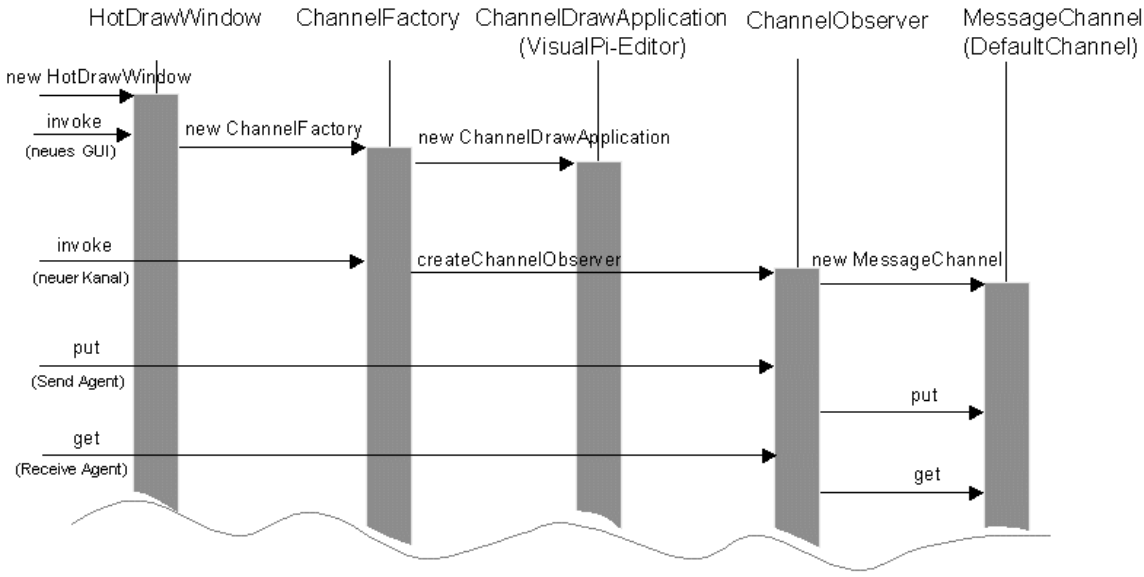

Abbildung 5.2: Reihenfolge der Objekterzeugung

5.3.2 Die Schnittstelle zu JHotDraw

Diese Schnittstelle basiert auf dem vom JHotDraw Framework zur Verfügung gestellten Rahmen, indem VisualPi die Klassen `StandardDrawingView` und `StandardDrawing` weiterhin verwendet. Das zentrale Element dieser Schnittstelle bildet die Klasse `ChannelDrawApplication`, die der VisualPi-Editor ist. Die Klasse `ChannelDrawApplication` (Abbildung 5.3) koordiniert die verschiedenen Komponenten, die zu einem Zeichnungs-Editor gehören, wobei sie z.B. für die Erzeugung

des Anzeigefensters, der Werkzeuge und Menus verantwortlich ist.

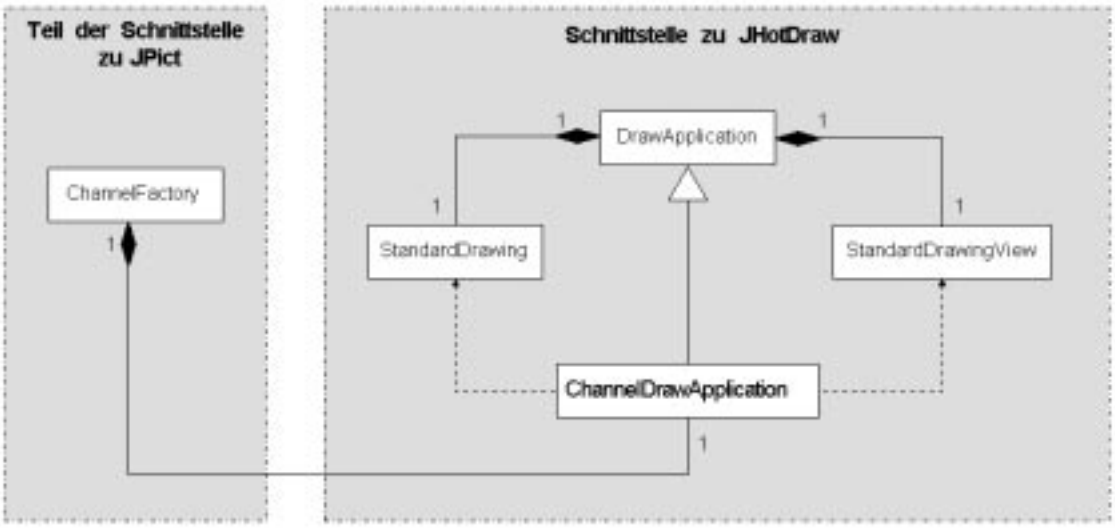

Abbildung 5.3: Die Schnittstelle zu JHotDraw

Das Anzeigefenster von VisualPi samt Werkzeugen und Menus ist der Abbildung 5.4 zu entnehmen. Dieses Anzeigefenster ist dem in der Abbildung 3.1 angezeigten Fenster sehr ähnlich, ausser der Werkzeugpalette und Menus, die zum Teil anders sind (eine ausführliche Beschreibung ist im Unterkapitel 6.2 zu finden).

`ChannelDrawApplication` ist eine Unterklasse der schon im Unterkapitel 3.2 vorgestellten Klasse `DrawApplication`, und es ist die Verantwortung von `ChannelFactory`, ein Objekt dieser Klasse zu erzeugen.

5.4 Erzeugung und Positionierung der Visualisierungsobjekte

5.4.1 Rolle der Visualisierungsobjekte

`ChannelObserver` ist das Visualisierungsobjekt, das in VisualPi einen Kanal bzw. seine Implementation `MessageChannel` aus dem JPict Framework darstellt. Die Beziehung zwischen `ChannelObserver` und dem Kanal wurde unter Verwendung des *Adapter* Patterns [10] implementiert. `ChannelObserver` spielt die *Adapter*-Rolle und `MessageChannel` die *Adaptee*-Rolle. Dafür enthält jedes `ChannelObserver`-Objekt ein `MessageChannel`-Objekt (Abbildung 5.1), und die Klasse `ChannelObserver` implementiert `put()` und `get()` der Schnittstelle `Channel`. Die zwei Methoden bilden eine Hülle (engl. *wrapper*) um die ursprünglichen Methoden `put()` und `get()` der Klasse `MessageChannel`. Agenten werden *put*- und *get*-Operationen auf `ChannelObservers` durchführen, die dann mittles *delegation* an die enthaltenen `MessageChannel`-Objekte weiterleiten.

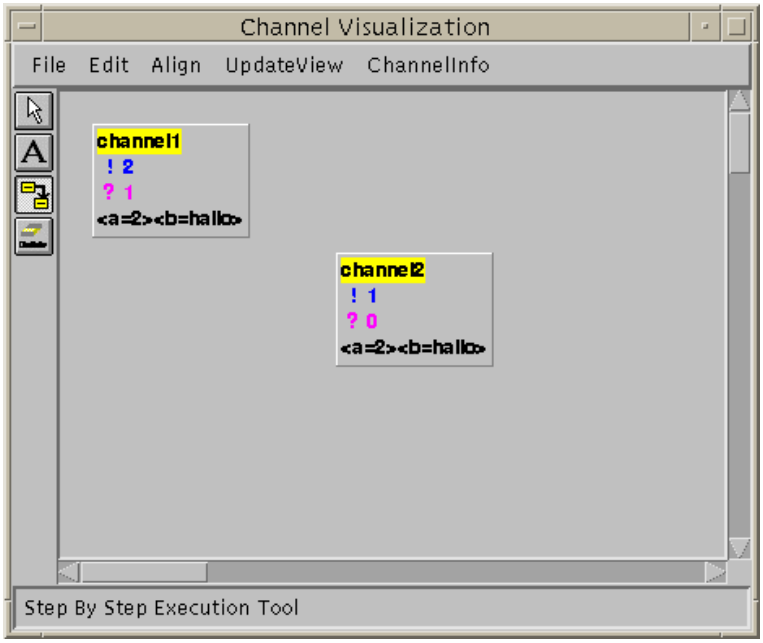

Abbildung 5.4: Der VisualPi-Editor

Weiterhin wird in dieser Arbeit der Begriff ”Kanal” anstelle von ”`ChannelObserver`-Objekt” verwendet, um die Lesbarkeit zu erhöhen.

5.4.2 Erzeugung von `ChannelObserver`

Jedes mal wenn `ChannelFactory` ein Objekt der Klasse `ChannelObserver` erzeugt, wird es auch im VisualPi-Editor dargestellt. Dafür wird ein Erzeugungswerkzeug gebraucht, names `ChannelCreationTool`, mit der Aufgabe ein `ChannelObserver`-Objekt zu erzeugen und darzustellen. Das `ChannelObserver`-Objekt ist als ein Rechteck dargestellt und enthält vier Felder, um folgende Informationen über den Kanal darzustellen (Abbildung 5.5):

1. Namen des darzustellenden Kanal.
2. Anzahl der *put*-Operationen. Das Ausrufezeichen vor der Zahl ist das Symbol für die *put*-Operation.
3. Anzahl der *get*-Operationen, die noch nicht durchgeführt worden sind. Das Fragezeichen vor der Zahl ist das Symbol für die *get*-Operation.
4. Den Wert (die Form), der gerade von einem Agent geholt wurde.

Die zwei Rechtecke in der Abbildung 5.4 stellen zwei `ChannelObserver`-Objekte dar. Die zwei Kanäle heissen ”channel 1” und ”channel 2”. Für ”channel 1” kann man daraus folgendes lesen: es wurden ”2” *put*-Operationen durchgeführt, eine *get*-Operation wartet auf die Ausführung und die

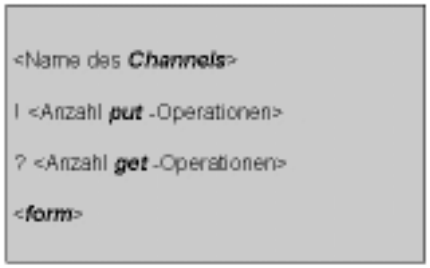

Abbildung 5.5: Aufbau eines `ChannelObservers`

letztgeholte Form ist ”<a=2><b=hallo>”.
Wenn die Formen zu lang und nicht gerade benötigt werden, kann man ihre Anzeige unterdrücken, durch Betätigung des vierten Werkzeuges von oben aus gesehen. Diese Fähigkeit ist in der Klasse `ChannelValueOffTool` eingebaut Abbildung 5.6.

Das Design dieses Modules ist der Abbildung 5.6 zu entnehmen. Die vier oben vorgestellten Felder sind vier Textobjekte, aus denen ein `ChannelObserver`-Objekt besteht. Dies wurde möglich, weil `ChannelObserver` als Unterklasse der `CompositeFigure`-Klasse von JHotDraw definiert wurde. `CompositeFigure` ermöglicht das Bilden eines Visualisierungsobjektes aus anderen Primitivobjekten (z.B. Textobjekte, Kreise, Linien etc), die JHotDraw zur Verfügung stellt. `ChannelCreationTool` und `ChannelValueOffTool` sind Unterklassen der abstrakten Klasse `AbstractTool` von JHotDraw. Die `ChannelObserver`-Objekte sind nach ihrer Erzeugung in `StandardDrawing` enthalten, wie alle Zeichnungsobjekte von JHotDraw.

5.4.3 Positionierung von `ChannelObserver`

Bevor ein Kanal im VisualPi-Anzeigefenster dargestellt wird, muss seine Position definiert werden. Die Position der Kanäle im Anzeigefenster wird vom π-Programm aus angegeben. Später können die Objekte von Benutzer interaktiv oder vom π-Programm aus verschoben oder gelöscht werden. Der Benutzer kann durch genaues Plazieren der `ChannelObserver`-Objekte im Anzeigefenster die Überlappung der Objekte vermeiden, das Verständnis des Programmablaufs erhöhen und einen besseren Überblick über die Erzeugung und Kommunikation zwischen den Kanälen bekommen.

Für die Positionierung wurde ein virtueller Raster auf das Anzeigefenster von VisualPi plaziert (Abbildung 5.7). Der Raster besteht aus Rasterzellen, die durch die Klasse `RasterCell` definiert sind. Jede Rasterzelle wird durch zwei Rasterkoordinaten, i (horizontal) und j (vertikal) angesprochen.

`ChannelDrawApplication` ist für den Raster verantwortlich, so dass jedesmal, wenn ein Kanal erzeugt und in das Anzeigefenster plaziert wird, die Rasterzelle, in der sich das Objekt befindet, als besetzt gekennzeichnet wird.

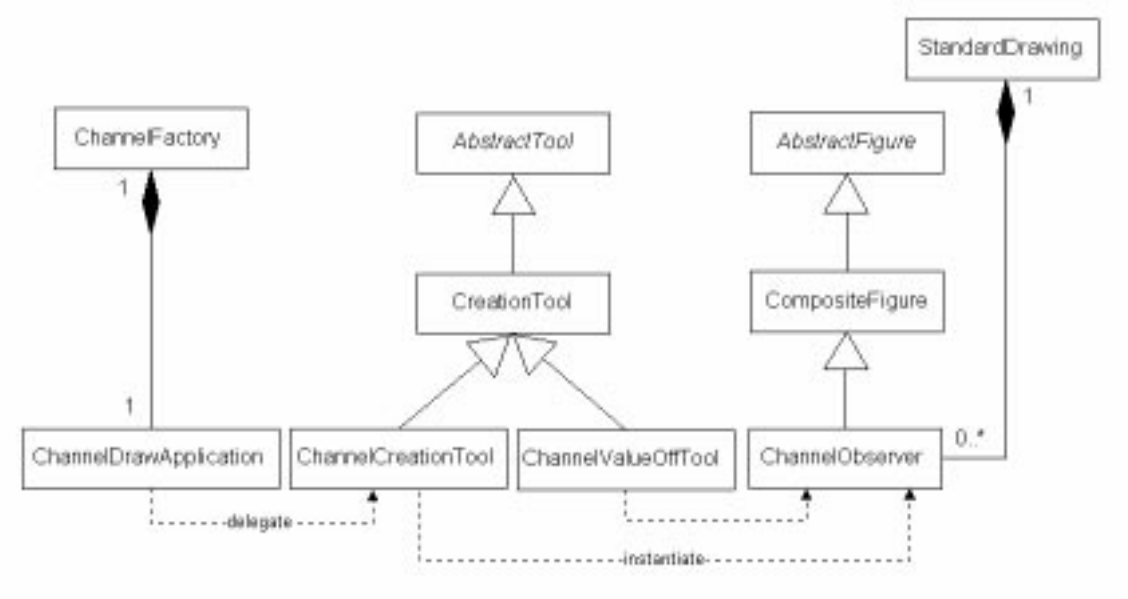

Abbildung 5.6: Erzeugung von `ChannelObserver`

Die Rasterzelle ist für die Umsetzung der i-, j-Rasterkoordinaten in die Pixel-Koordinaten, x und y, der linken, oberen Ecke des `ChannelObserver`-Objektes, verantwortlich (Abbildung 5.7). `RasterCell` enthält ein `Dimension` Objekt, das die Pixel-Koordinaten, x (horizontal) und y (vertikal) einkapselt (Abbildung 5.8). Somit kennt das Werkzeug `ChannelCreationTool`, das die Kanäle erzeugt, das ganze Rasterungsverfahren und `RasterCell` nicht, sondern es arbeitet direkt mit den Pixel-Koordinaten x und y.

5.5 Anzeige der Kanalzustände

Ein Kanal kann sich in einem von den im Unterkapitel 2.2 beschriebenen drei Zuständen befinden: *nonempty, empty* oder *blocking*. Diese Zustände werden durch das Verhältnis zwischen der Anzahl der *put*- und derjenigen der *get*-Operationen definiert. Weil `ChannelObserver`-Objekte diese zwei Zahlen anzeigen (Abbildung 5.5), ist es möglich für den Benutzer in jedem Moment den Zustand eines Kanals abzulesen.

Der wichtigste Kanalzustand ist der *nonempty*-Zustand, in dem die Wertwarteschlange Formen enthält und Agenten darauf warten *get*-Operationen ausgeführen. In diesem Fall ist die Anzahl der *put*-Operationen grösser als oder gleich derjenigen der *get*-Operationen, und beide Zahlen sind grösser als Null.

Dieser Zustand wird durch eine spezielle Animation hervorgehoben, nämlich das Blinken des Kanalnamen des `ChannelObserver`-Objektes. Dafür ist `ChannelDrawApplication` verantwortlich, indem es ständig den Zustand jedes Kanals nachfragt.

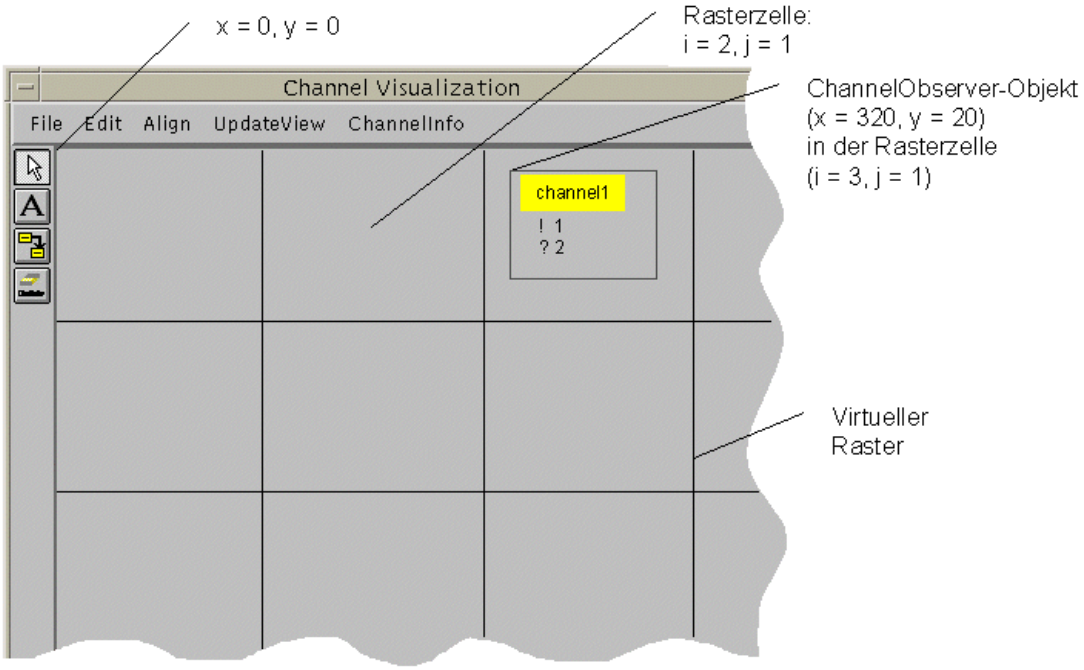

Abbildung 5.7: Abbildung zwischen den Raster- und den Pixel-Koordinaten

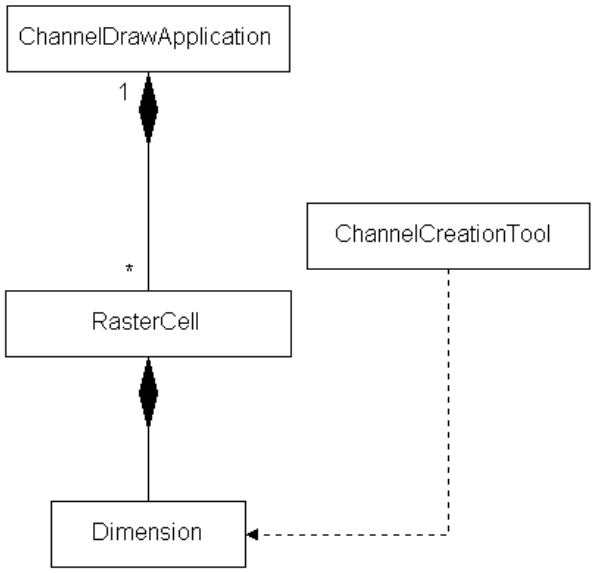

Abbildung 5.8: Positionierung von `ChannelObserver`

5.6 Interaktive Ausführung eines π-Programmes

Interaktive Ausführung bedeutet für VisualPi, dass der Benutzer, mittels Mausklick, ein π-Programm schrittweise ausführen kann. Das Programm kann ausgeführt werden, wenn der Benutzer das Werkzeug ”Step By Step Execution Tool” (Abbildung 5.4 - dritter Knopf in der Werkzeugpalette) betätigt und anschliessend auf Kanäle klickt.

Es gibt einen einzigen Kanalzustand, in dem der Mausklick des Benutzers Wirkung zeigt, nämlich wenn der Kanalname blinkt (Abschnitt 5.5). Klickt der Benutzer auf dem Kanal, so wird ein weiterer Schritt des π-Programmes ausgeführt. Was genau in einem Ausführungsschritt passiert, wird anhand eines Beispiels vorgestellt, das auf dem im Abschnitt 2.1 erklärten Beispiel beruht und dieselben zwei Kanäle enthält:

```
...
channel1 ! <a = 1>
|
channel1 ? X > {channel2 ! <a = 10>
                |
                channel2 ? Y > channel1 ! <b = 100>}
```

Die Zeile ”channel1 ! <a = 1>” stellt einen *Send Agent* dar, der eine *put*-Operation auf den ”channel1” ausführt, indem die Form ”<a = 1>” in die Wertwarteschlange des ”channel1” gelegt wird.

Die Zeilen ”channel1 ? X > {...}” stellen einen *Receiving Agent* dar, der eine *get*-Operation auf den ”channel1” ausführt. Der Code dieser Operation enthält zwei Teile:

1. vor dem Zeichen ”>”: ”channel1 ? X” bedeutet, dass die Form ”<a = 1>” aus der Wertwarteschlange des ”channel1” geholt wird und mit dem Bezeichner ”X” verbunden wird.

2. nach dem Zeichen ”>”: Dieser Teil wird ausgeführt nachdem der erste Teil ausgeführt wurde, d.h., nachdem die Form ”<a = 1>” mit dem Bezeichner X verbunden wird. Wenn man den Code zwischen den geschweiften Klammern anschaut enthält er nichts anderes als eine neue *put*- und *get*-Operation, die auf ”channel2” ausgeführt werden. Das Zeichen ”|” deutet darauf hin, dass die zwei Operationen parallel ausgeführt werden. Das Programmstück zwischen den geschweiften Klammern bedeutet, dass eine *put*-Operation auf ”channel2” ausgeführt wird, indem die Form ”<a = 10>” in die Wertwarteschlange des ”channel2” gelegt wird. Parallel dazu wird eine *get*-Operation auf demselben Kanal ausgeführt, indem die Form ”<a = 10>” aus der Wertwarteschlange des ”channel2” geholt wird und mit dem Bezeichner ”Y” verbunden wird, und anschliessend wird die *put*-Operation ”channel1 ! <b = 100>” auf ”channel1” durchgeführt.

In einem Ausführungsschritt werden alle zu einem Zeitpunkt parallel ausführbaren *put*-Operationen gezählt und ausgeführt und alle *get*-Operationen nur gezählt aber nicht ausgeführt. Der *put*- und *get*-Zähler (Abschnitt 5.4.2) der dargestellten Kanäle, auf die Operationen von Agenten ausgeführt werden, werden entsprechend inkrementiert. Das obige Beispiel wird in drei Schritten ausgeführt:

1. Im ersten Schritt, nach dem Starten des π-Programmes, wird die *put*-Operation ”channel1 ! <a = 1>” ausgeführt und die *get*-Operation ”channel1 ? X > {...}” nur gezählt.

Beide Zähler von ”channel1” werden um eins inkrementiert, und infolge dessen blinkt ”channel1”.

2. Klickt man auf ”channel1” in einem zweiten Schritt, so wird die *get*-Operation ”channel1 ? X > {...}” ausgeführt. Im zweiten Teil dieser *get*-Operation wird die *put*-Operation ”channel2 ! <a = 10>” ausgeführt und die *get*-Operation ”channel2 ? Y > channel1 ! <b = 100>” nur gezählt. Beide Zähler von ”channel2” werden um eins inkrementiert und ”channel2” blinkt. Weil zu diesem Zeitpunkt die *get*-Operation auf den ”channel1” ausgeführt wurde, wird der Zähler für die *get*-Operationen um eins dekrementiert.

3. Klickt man auf ”channel2”, so wird in einem letzten Schritt die *get*-Operation ”channel2 ? Y > channel1 ! <b = 100>” ausgeführt, indem die *put*-Operation ”channel1 ! <b = 100>” auf ”channel1” ausgeführt wird, wobei nur der Zähler für die *put*-Operationen um eins inkrementiert wird.

Jedes Ergebnis einer *put*- und *get*-Operation wird angezeigt. Das Ergebnis jeder *get*-Operation erscheint im vierten Feld, ”<form>”, eines Kanals (Abbildung 5.5) und dasjenige jeder *put*-Operation in einem separaten Fenster (Abschnitt 5.7).

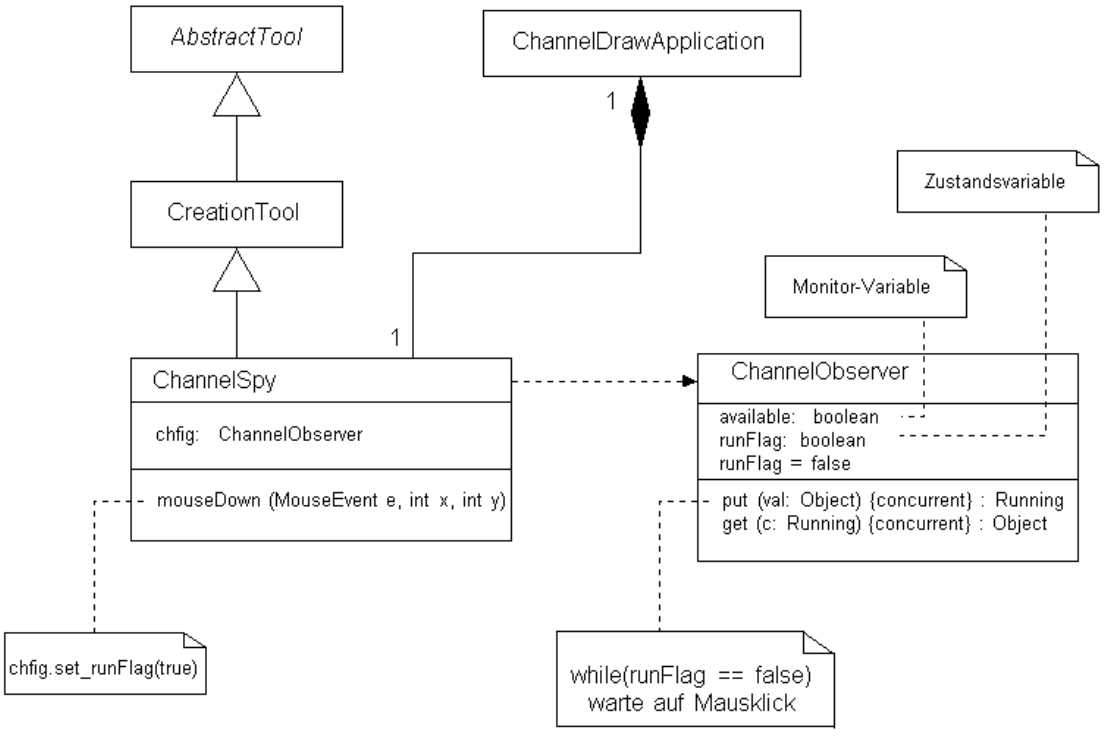

Abbildung 5.9: Interaktive Ausführung eines π-Programmes

Diese Fähigkeit ist durch die Zusammenarbeit (engl. *collaboration*) von zwei Klassen einge-

baut, `ChannelSpy` und `ChannelObserver` (Abbildung 5.9). Die schrittweise Ausführung eines π-Programmes wird ermöglicht durch:

- Definition der `put`- und `get`-Methoden in der Klasse `ChannelObserver` als *synchronized*.
- Verwendung der Monitor-Variable `available`.
- Verwendung der Zustandsvariable `runFlag`.

Der Zustand der `runFlag`-Variable wird durch Mausklick geändert. Somit ist es gestattet durch ständiges Abfragen des `runFlag`-Zustandes innerhalb der `put`-Methode die Ausführung des π-Programmes fortzusetzen oder weiterhin auf den Mausklick abzuwarten (Abbildung 5.9).

`ChannelSpy` ist indirekt als Unterklasse von `Tool` definiert (Abbildung 5.9). Somit ist `ChannelSpy` durch das ”Step By Step Execution Tool” im VisualPi-Anzeigefenster vertreten. Wenn dieses Werkzeug selektiert wird, werden die Mausklicks von der `mouseDown(MouseEvent e, int x, int y)`-Methode behandelt, die in `ChannelSpy` implementiert wurde.

5.7 Anzeige der Kanal-Wertwarteschlange

Während der Ausführung eines π-Programmes ist nicht nur die Darstellung der Kanalzustände wichtig, sondern auch die Anzeige der Formen, die sich in der Wertwarteschlange jedes Kanals befinden.

Bei jeder *put*-Operation wird eine Form in die Wertwarteschlange gelegt, und bei jeder *get*-Operation wird die erste Form darausgelesen, denn die Wertwarteschlange ist vom Typ FIFO. Die Anzeige der Kanal-Wertwarteschlange folgt in einem separaten Fenster (das untere Fenster mit dem Namen ”Info” in der Abbildung 5.10), um die Übersicht über die erzeugten Kanäle im VisualPi-Anzeigefenster nicht zu vermindern.

Diese Fähigkeit ist im VisualPi-Editor durch ein Menu verfügbar, names ”ChannelInfo”. Das Menu enthält zwei Einträge, ”InfoOn” und ”InfoOff”. Durch Klicken auf ”InfoOn” wird das Info-Fenster geöffnet, in dem reihenweise der Name jedes selektierten Kanals und die Formen in der Wertwarteschlange angezeigt werden (Abbildung 5.10).

Während das ”Info”-Fenster geöffnet ist und über das Werkzeug ”Step By Step Execution Tool” die interaktive Ausführung des π-Programmes fortgesetzt wird, wird der Inhalt des Fensters immer aufgefrischt, so dass die aktuellen Formen in der Kanal-Wertwarteschlange angezeigt werden. Durch Klicken auf ”InfoOff” wird das Fenster zugemacht.

Das Design dieses Modules ist in der Abbildung 5.11 vorgestellt. Der Kern von VisualPi, `ChannelDrawApplication`, trägt die Verantwortung für die Anzeige der Kanal-Wertwarteschlange.

Das Menu wird von der Klasse `CommandMenu` von JHotDraw und die Menueinträge werden von der Klasse `MenuItem` zur Verfügung gestellt. Die Menueinträge ”InfoOn”- und ”InfoOff”sind je mit

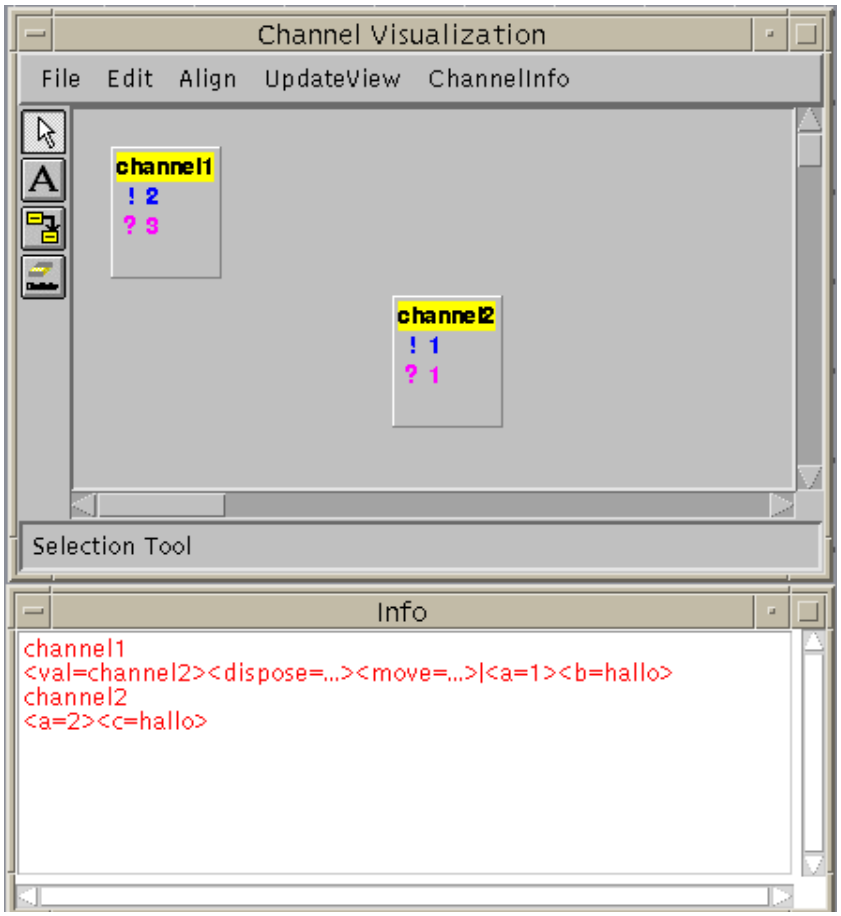

Abbildung 5.10: Das ”Info”-Fenster, das die Kanal-Wertwarteschlangen anzeigt

einem Befehlblock verbunden. Die zwei Befehlblöcke sind in der Klasse `InfoOnCommand`, bzw. `InfoOffCommand`, in der Methode `execute()` beinhaltet. Beide Klassen sind Unterklassen der Klasse `Command` von JHotDraw. Das Fenster, in dem die Kanal-Wertwarteschlangen angezeigt werden, ist in der Klasse `InfoWindow` implementiert, die eine Unterklasse von `Frame` ist. Für die Anzeige enthält `InfoWindow` noch ein `TextArea`-Objekt.

5.8 Dienste

VisualPi ist ein interaktives Visualisierungswerkzeug. Interaktiv bedeutet auf der einen Seite Aktionen von Benutzer durch Mausklick im Anzeigefenster vom VisualPi betätigen und auf der anderen Seite verschiedene Aktionen vom π-Programm aus auslösen. Solche Aktionen wurden weiter oben in Abschnitt 5.4.3 vorgestellt, indem der Benutzer die Kanäle selbst vom π-Programm aus auf dem VisualPi-Anzeigefenster plazieren konnte. Der Benutzer kann noch zwei weitere Aktionen aus einem π-Programm durchführen:

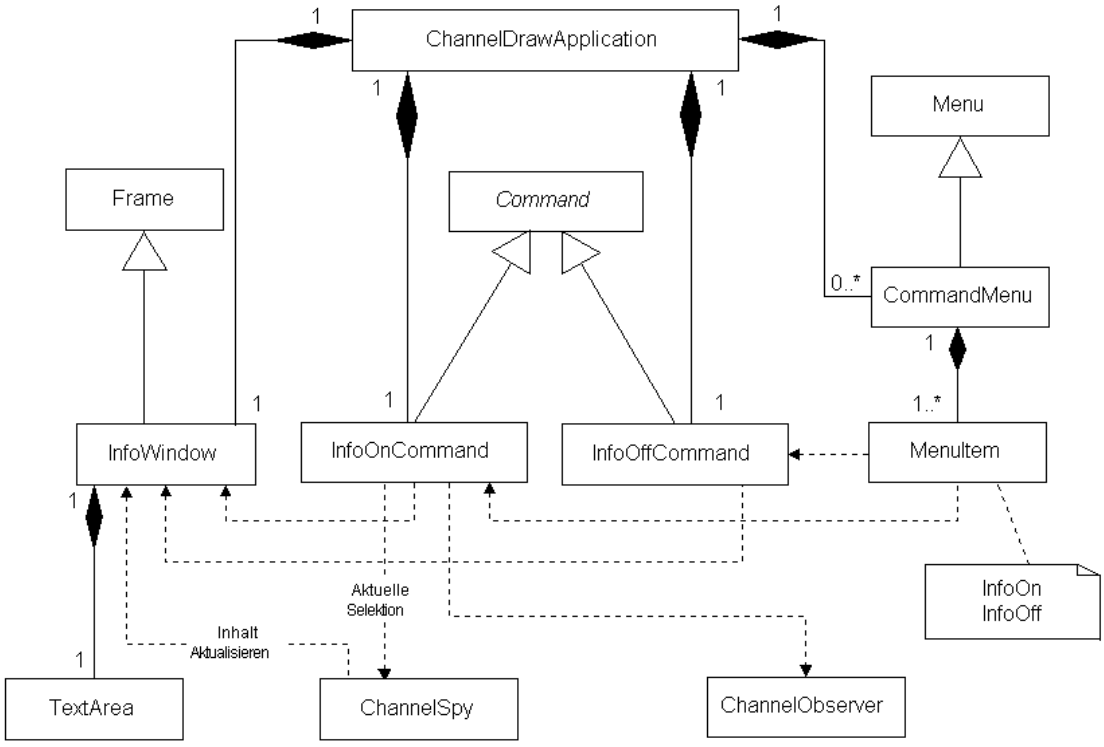

Abbildung 5.11: Anzeige der Kanal-Wertwarteschlange

- Verschieben von `ChannelObserver`-Objekten.
- Löschen von `ChannelObsever`-Objekten, ohne dass auch der entsprechende Kanal mitgelöscht wird.

Diese zwei Aktivitäten wurden als ”Dienste” (*services*) innerhalb von zwei Klassen, `MoveService` und `DisposeService`, eingeführt. Beide Klassen sind Unterklassen der abstrakten Klasse `AbstractService` von JPict und definieren ihren Dienst, Verschieben oder Löschen, in der Methode `invoke(Form f)`.

Für die Dienste ist `ChannelFactory` verantwortlich, so dass jedem Kanal bei seiner Erzeugung zwei eigene Dienste zugewiesen werden. Das zugehörige Klassendiagramm ist in der Abbildung 5.12 dargestellt.

5.9 Anzeige des Nachrichtenaustausches zwischen den Kanälen

Im Abschnitt 5.6 wurde ausführlich erklärt, was genau bei der Ausführung einer *put*-Operation und einer *get*-Operation auf einen Kanal passiert. Wichtig für diesen Abschnitt ist auch der zweiteilige Aufbau einer *get*-Operation. Als Nachrichten, die zwischen den Kanälen ausgetauscht werden, werden

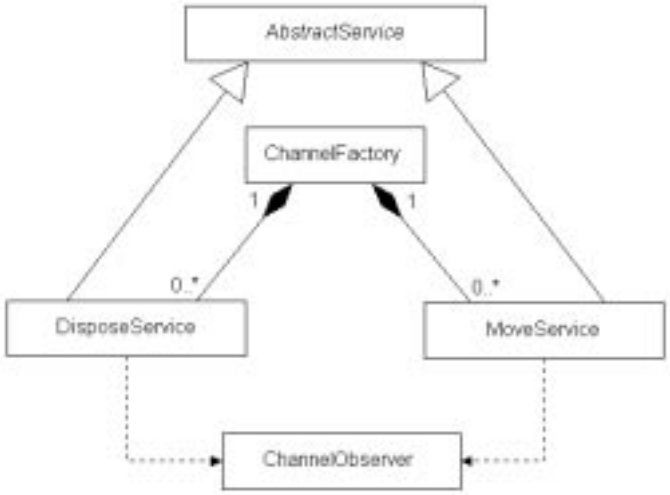

Abbildung 5.12: Dienste von VisualPi

die *put*- und *get*-Operationen auf Kanäle bezeichnet, die im zweiten Teil einer *get*-Operation (Abschnitt 5.6) aufgelistet werden. Wenn ein Kanalname blinkt und der Benutzer darauf klickt, werden Verbindungslinien zwischen dem entsprechenden Kanal, Quellkanal, und den Kanälen angezeigt, auf welchen *put*- oder/und *get*-Operationen aus dem zweiten Teil der ausführenden *get*-Operation sich beziehen. Jede Verbindungslinie ist mit einem Pfeil versehen, am entgegengesetzten Ende zum Quellkanal. Wird auf einen anderen Kanal geklickt, so werden neue Verbindungslinien angezeigt, und die alten werden gelöscht.

Das folgende Beispiel, das eine vereinfachte Version des im Abschnitt 5.6 vorgestellten Beispiels ist, soll verdeutlichen wie die Anzeige des Nachrichtenaustausches zwischen den Kanälen funktioniert:

```
...
channel1 ! <a = 1>
|
channel1 ? X > channel2 ! <a = 10>
```

Die zweite Zeile stellt ein *Send Agent* dar, der eine *put*-Operation auf den ”channel1” ausführt. Die vierte Zeile stellt ein *Receiving Agent* dar, der eine *get*-Operation auf den ”channel1” ausführt, indem die Form ”<a = 1>” aus der Wertwarteschlange des ”channel1” geholt wird und mit dem Bezeichner ”X” verbunden wird. Anschliessend wird die *put*-Operation ”channel2 ! <a = 10>” auf ”channel2” ausgeführt wird, indem die Form ”<a = 10>” in die Wertwarteschlange des ”channel2” gelegt wird. Das Ergebnis der Ausführung des obigen Beispiels wird in der Abbildung 5.13 angezeigt. Es wird eine Verbindunslinie zwischen ”channel1” und ”channel2” angezeigt. ”channel1” ist der Quellkanal. Im ”Info”-Fenster derselben Abbildung wird die Form ”< a = 10 >” in der Wertwarteschlange von ”channel2” angezeigt.

Das Design dieser Fähigkeit ist der Abbildung 5.14 zu entnehmen. Die Verbindunslinien werden durch

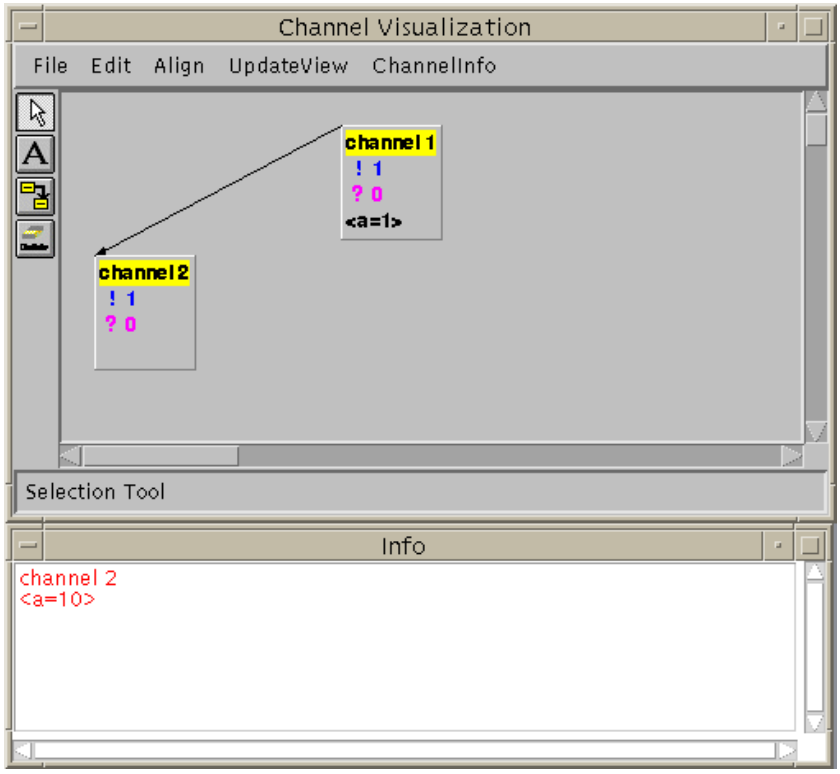

Abbildung 5.13: Anzeige der Nachrichten zwischen den Kanälen durch Verbindungslinien

die Klasse `MessageExchange` dargestellt. `StandardDrawing` enthält die Verbindungslinien, wie alle Zeichnungsobjekte von JHotDraw.

Beim Klicken auf einen blinkenden Kanal wird er als ”lastClickedChannel” gekennzeichnet. Nach der Ausführung einer *put*- oder *get*-Operation auf einen Kanal, wird dieser die Verbindungslinie zum ”last-ClickedChannel”zeichnen.

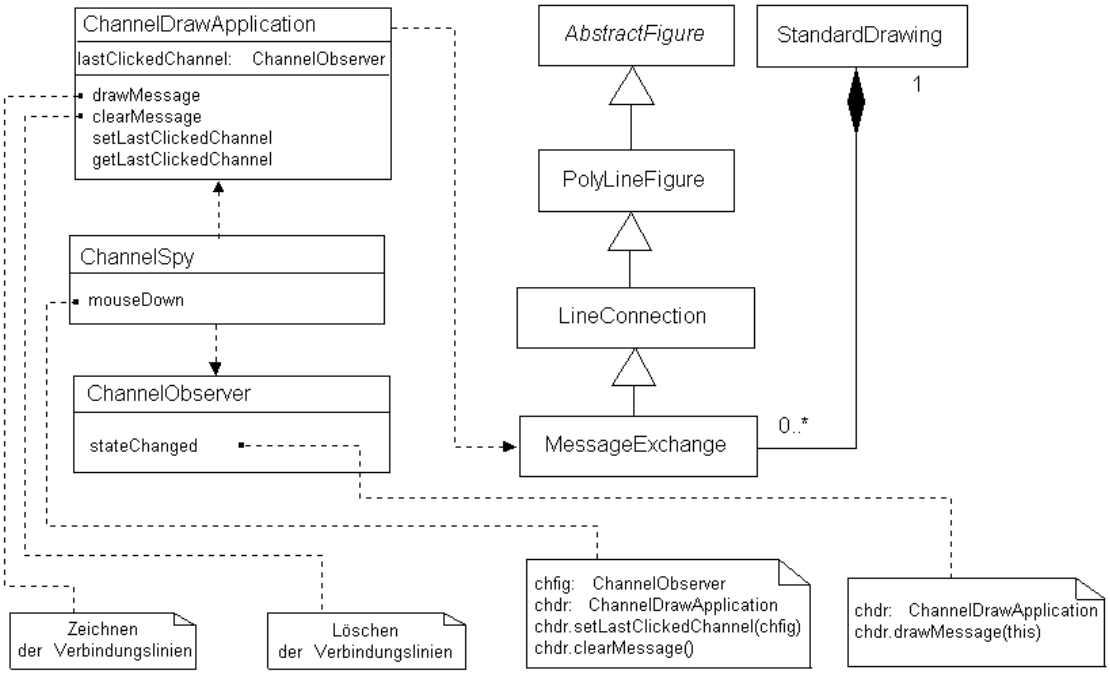

Abbildung 5.14: Anzeige des Nachrichtenaustausches zwischen den Kanälen

Kapitel 6

Implementation von VisualPi

Dieses Kapitel baut auf dem vorherigen Kapitel auf und stellt kurz die Implementation des Visualisierungswerkzeuges, VisualPi, mittels Codebeispiele vor. Die Unterkapiteln entsprechen den im Kapitel 5 vorgestellten Unterkapiteln.

6.1 Implementation der Schnittstelle zu JPict

Wie schon im Unterkapitel 5.3.1 beschrieben, besteht diese Schnittstelle aus folgenden Klassen: `HotDrawWindow`, `ChannelFactory`, `ChannelObserver` und `MessageChannel`. Die Implementation der Schnittstelle zu JPict und die Einbindung in PiL von VisualPi werden anhand eines kurzen Beispiels vorgestellt. Das Beispiel enthält dasselbe π-Programm, das im Unterkapitel 2.1 beschrieben wurde, mit entsprechenden Änderungen und zusätzlichen Befehlen aus der hier beschriebenen Schnittstelle. Das geänderte π-Programm ist im rechten Fenster der Abbildung 6.1 zu finden.

Weiter unten werden nur die Unterschiede zum urspünglichen Beispiel (Abbildung 2.1) beschrieben. Die erste Zeile

```
extern "HotDrawWindow" window
```

erzeugt ein `HotDrawWindow`-Objekt mit dem Namen ”window”. Der Befehl

```
let Channel_Factory := window(<x = 3> <y = 3> <message = "yes">) ;
```

führt die Funktion

```
window(<x = 3> <y = 3> <message = "yes">)
```

aus, die ein Objekt der Klasse `ChannelFactory` zurückgibt, names ”Channel_Factory”. Dabei wird der VisualPi-Editor (GUI) angezeigt. Der Aufruf dieser Funktion entspricht der Ausführung der `invoke(Form f)`-Methode der Klasse `HotDrawWindow`:

```
protected Object invoke(Form f){
```

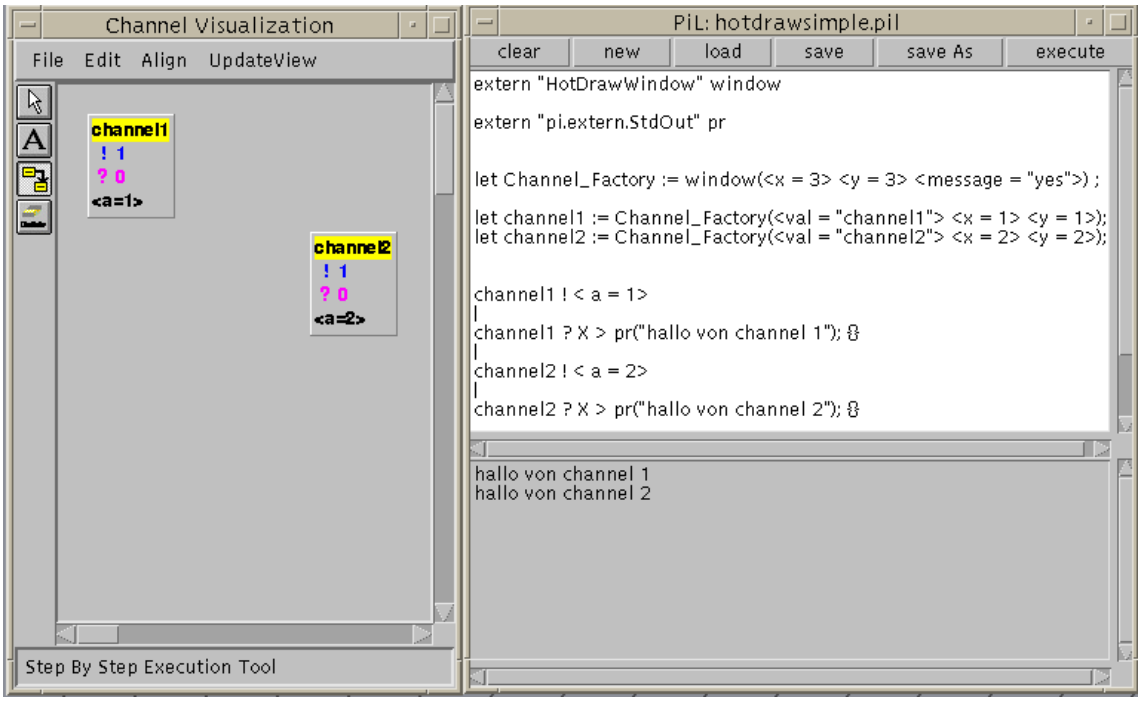

Abbildung 6.1: Rechts: das PiL-Eingabefenster, links: Der VisualPi-Editor

```
  //Umwandlung des x- und y-Wertes in entsprechende Zahlen (x_value, y_value)
  ...

  ChannelFactory cf = new ChannelFactory(x_value, y_value);

  return cf;
}
```

Das Argument ”<x = 3><y = 3> <message = ”yes”>” ist eine Form, das bedeutet:

- der Teil ”<x = 3><y = 3>” gibt die Grösse des virtuellen Rasters des VisualPi-Anzeigefensters an. In diesem Beispiel besteht der Raster aus 3x3 Rasterzellen.
- der restliche Teil ”<message = ”yes”>” ist ein Schalter mit ”yes” und ”no” als mögliche Werte. Ist der Wert ”yes”, so werden die Verbindunslinien für den Nachrichtenaustausch zwischen den Kanälen angezeigt. Ist der Schalter auf ”no” gesetzt, so werden keine Verbindunslinien dargestellt, und die Ausführung des π-Programmes läuft schneller. Wenn dieser Teil in der obigen Form fehlt, wird automatisch der Schalter auf ”no” gesetzt.

In den nächsten zwei Zeilen

```
let channel1 := Channel_Factory(<val ="channel1"> <x = 1> <y = 1>);
let channel2 := Channel_Factory(<val ="channel2"> <x = 2> <y = 2>);
```

werden zwei `ChannelObserver`-Objekte erzeugt, ”channel1” und ”channel2”, die im linken Fenster der Abbildung 6.1, zu sehen sind. Die zwei obigen Zeilen, die mit ”let” anfangen, ersetzen die Zeilen, die mit ”new” im ursprünglichen π-Programm anfingen (Abbildung 2.1), denn durch die Erzeugung der zwei `ChannelObserver`-Objekte werden auch zwei `MessageChannel`-Objekte miterzeugt. Somit bleiben die *put*- und *get*-Operationen aus dem restlichen Teil des π-Programmes unverändert.

Durch die Aufrufe ”Channel_Factory(...)” wird die `invoke(Form f)`-Methode der Klasse `ChannelFactory` zwei mal ausgeführt:

```
protected Object invoke(Form f){
  //Wandle die x- und y-Wertes in Zahlen um

  try{
    //Erzeuge ein ChannelObserver-Objekt
    ChannelObserver chfig = (ChannelObserver) chframe.createChannelObserver(...)
    ...
    //Erzeuge einen Loeschen-Dienst
    DisposeService ds = new DisposeService(...);
    //Erzeuge einen Verschieben-Dienst
    MoveService ms = new MoveService(...);
    ...
  }catch (ArrayIndexOutOfBoundsException e){
    //Fehlermeldungen wenn die Rastergroesse ueberschritten wird
  }catch (RasterCellOccupiedException e){
    //Fehlermeldungen wenn die Rasterzelle besetzt ist
}}
  return ... //Eine Form wird zurueckgegeben
}
```

Der erste Teil des Argumentes ”<val = ”channel1”>” legt den Namen des Kanals fest. Der restliche Teil ”<x = 1> <y = 1>” gibt die Position des Kanals im virtuellen Raster des VisualPi-Anzeigefensters an, wobei folgende Bedingungen eingehalten werden müssen:

$1 \leq x \leq x\text{-}Koordinate\ \ des\ \ Rasters$ und $1 \leq y \leq y\text{-}Koordinate\ \ des\ \ Rasters$.

Somit kann der Benutzer selbst die Kanäle positionieren (Abschnitt 5.4.3). Der Kanal mit dem Namen ”channel1” wird z.B. oben links, in der ersten Spalte und ersten Zeile des virtuellen Rasters plaziert (Abbildung 6.1, linkes Fenster).

6.2 Implementation der Schnittstelle zu JHotDraw

Die Implementation dieser Schnittstelle wird anhand des im Abschnitt 6.1 angegeben Beispiels vorgestellt. Im vorherigen Abschnitt wurde erklärt, wie bei der Erzeugung des `ChannelFactory`-Objektes "Channel_Factory" die `invoke(Form f)`-Methode der Klasse `HotDrawWindow` ausgeführt wird. Diese Methode kreiert durch Aufruf des Konstruktors der `ChannelFactory`-Klasse ein `ChannelFactory`-Objekt:

```
public class ChannelFactory extends Function implements Channel {

  ChannelDrawApplication chframe = null;

  public ChannelFactory(int x, int y){
    super();
    if (chframe == null){
      chframe = new ChannelDrawApplication(x, y);
      chframe.open();
    }
  }
...
}
```

Die Anweisung

```
chframe = new ChannelDrawApplication(x, y);
```

erzeugt das `ChannelDrawApplication`-Objekt, den VisualPi-Editor. Somit wird bei der Erzeugung des `ChannelFactory`-Objektes auch das `ChannelDrawApplication`-Objekt mitkreiert. Der Befehl

```
chframe.open();
```

kreiert und initialisiert alle Komponenten des VisualPi-Editors. Der Editor ist im linken Fenster der Abbildung 6.1 dargestellt und besteht aus:

- Anzeigefenster "Channel Visualization": enthält die `ChannelObserver`-Objekte, die während der Ausführung eines π-Programmes erzeugt werden.
- Werkzeugpalette - links neben dem Anzeigefenster: enthält folgende vier Werkzeuge:
  - "Selection Tool": selektiert ein oder, durch gleichzeitiges Drücken der SHIFT-Taste, mehrere `ChannelObserver`-Objekte.
  - "Text Tool": fügt ein Text-Objekt ein.
  - "Step By Step Execution Tool": wenn dieses Werkzeug aktiv und auf ein `ChannelObserver`-Objekt geklickt wird, wird die schrittweise Ausführung eines π-Programmes ermöglicht (Abschnitt 5.6).

  - ”Disable Get Information Tool”: wenn dieses Werkzeug aktiv und auf ein `ChannelObserver`-Objekt geklickt wird, wird die Information im vierten Feld des entsprechenden `ChannelObserver`-Objektes unterdrückt (Abschnitt 5.4.2).

- ”File”-Menu: enthält die üblichen Datei-Operationen, wie ”Save As...”, die die Zeichnung in eine Datei mit der Endung ”draw” abspeichert, oder ”Open...”, die eine existierende Datei öffnet, oder ”Print...”, die eine Zeichnung ausdruckt.
- ”Edit”-Menu: besteht aus Einträgen wie ”Cut”, ”Copy”, ”Paste” oder ”Delete”. Alle Operationen beziehen sich auf die Zeichnungsobjekte, die im Anzeigefenster enthalten sind.
- ”Align”-Menu: Befehle zum Ausrichten der Zeichnungsobjekte im Anzeigefenster, z.B., ”Left”, ”Right” oder ”Zenter”.
- ”UpdateView”-Menu: die zwei Menueinträge bewirken eine Auffrischung des Anzeigefensters.
- ”ChannelInfo”-Menu: besteht aus den Einträgen ”InfoOn” und ”InfoOff”. ”InfoOn” wird bei der Selektion von mindestens einem Kanal aktiv. Wird ”InfoOn” gedrückt, so wird das ”Info”-Fenster mit den Informationen über die Namen und Wertwarteschlangen der selektierten Objekte angezeigt. Wird auf ”InfoOff” gedrückt, so wird das ”Info”-Fenster geschlossen.

6.3 Implementation der Erzeugung der Visualisierungsobjekte

Wie schon im Abschnitt 6.1 beschrieben, wird ein `ChannelObserver`-Objekt während der Ausführung der `invoke(Form f)`-Methode der Klasse `ChannelFactory` erzeugt. Dies wird durch *delegation* der Erzeugungsoperation an das Werkzeug `ChannelCreationTool` realisiert, das auf die Erzeugung und Darstellung von `ChannelObserver`-Objekten spezialisiert ist:

```
public class ChannelCreationTool extends CreationTool {
...

  protected Figure createFigure(String name, InfoWindow infow) {

    ChannelObserver chFig = new ChannelObserver(view(), name, infow);
    return chFig;
  }
...

}
```

Bei der Erzeugung eines `ChannelObserver`-Objektes wird auch ein `MessageChannel` miterzeugt:

```
public class ChannelObserver extends CompositeFigure implements Channel{

  private TextFigure name;
  private TextFigure Channel_put;
  private TextFigure Channel_get;
  private TextFigure Channel_getValue;

  private MessageChannel ch_;

  public ChannelObserver(DrawingView view, String Name, InfoWindow infow) {
    ...
    initialize();
    ch_ = new MessageChannel(Name);
  }
...
}
```

Die ersten vier Zeilen beinhalten die Deklarationen der vier Textobjekte, aus denen ein `ChannelObserver`-Objekt besteht. Die Methode `initialize()` instantiert und initialisiert die vier Textobjekte. Die Klasse `TextFigure` gehört zu JHotDraw und ist eine Unterklasse der abstrakten Klasse `AbstractFigure`.

Die `put`- und `get`-Methoden eines `ChannelObserver`-Objektes bilden nur eine Hülle (engl. *wrapper*) um die urspünglichen `put`- und `get`-Methoden eines `MessageChannel`-Objektes. Somit ist es gestattet zusätzliche Operationen auszuführen, wie z.B. Aktualisieren der vier Felder des `ChannelObserver`-Objektes (Abschnitt 5.4.2):

```
public synchronized Object get(Running c){
...
  //Aktualisieren der Inhalte der TextFigure-Objekte
...
  Object r = ch_.get(c); //Delegation an MessageChannel
...
  return r;
}

public synchronized Running put(Object val) throws Exception {
...
  //Aktualisieren der Inhalte der TextFigure-Objekte
...
  Running r = ch_.put(val); //Delegation an MessageChannel
...
  return r;
}
```

6.4 Implementation der Anzeige der Kanalzustände

Wie schon im Unterkapitel 5.5 beschrieben wurde, gibt es einen einzigen Kanalzustand, in dem, wenn das Werkzeug ”Step By Step Execution Tool” aktiviert ist, der Mausklick des Benutzers auf den entsprechenden Kanal Wirkung zeigt. Dieser Zustand wurde hervorgehoben durch Blinken des Kanalnamen. Das Blinken ist in einem *thread* der Klasse `ChannelDrawApplication` eingebaut:

```
public class ChannelDrawApplication extends DrawApplication implements Runnable{
...
  private Thread ChannelBlinker;
  private boolean ThreadIsRunning = false;

  public void start(){
    ChannelBlinker = new Thread(this);
    ThreadIsRunning = true;
    ChannelBlinker.start();
  }

...
  public void run(){
    while (ThreadIsRunning){
      //Ueberpruefe das ChannelBlinks-Flag jedes ChannelObserver-Objektes:
      // if ChannelBlinks == true (der spezielle Channel-Zustand tritt auf)
      //    starte das Blinken des ChannelObserver-Namen

    }
  }
...
}
```

Der Thread wird einmal beim Erzeugen des ersten Kanals mitkreiert und existiert solange das `ChannelDrawApplication`-Objekt existiert.

6.5 Implementation der Dienste

Im Abschnitt 5.8 wurden die zwei Dienste vorgestellt, die von einem π-Programm aus ausgeführt werden können: Verschieben und Löschen von `ChannelObserver`-Objekten.

Verschieben von `ChannelObserver`-Objekten

Folgender Befehl in einem π-Programm bewirkt das Verschieben eines `ChannelObserver`-Objektes:

```
channel_name.move(<x = Zahl1> <y = Zahl2>)
```

wobei das Argument ”<x = Zahl1> <y = Zahl2>” die neue absolute Position des `ChannelObserver`-Objektes im virtuellen Raster des VisualPi-Anzeigefensters angibt. Der

obige Aufruf entspricht der Ausführung der `invoke(Form f)`-Methode der Klasse `MoveService` (Abbildung 5.12):

```
public class MoveService extends AbstractService {

  private DrawingView fView_;
  private ChannelObserver chfig_;
  private RasterCell rc_old;
  private ChannelDrawApplication chframe_;

...

  protected Form invoke(Form f) {

    //Wandle die x- und y-Werte in entsprechende Zahlen (x_value, y_value) um

    try{
      //Setze die alte Rasterzelle, in der sich der Kanal befand, als frei
      rc_old.setCell_notOccupied();
      ...

      //Setze die neue Rasterzelle (Position im Raster: x_value, y_value),
      //in der sich der Kanal befindet, als besetzt
      RasterCell rc_new = chframe_.getRasterCell(x_value, y_value);
      rc_new.setCell_occupied();
      ...

      //Loesche das ChannelObserver-Objekt an der alten Position und zeichne es
      //an der neuen Position
      chfig_.displayBox(fAnchorPoint, fAnchorPoint);

    } catch (ArrayIndexOutOfBoundsException e){
      //Fehlermeldungen wenn die Rastergroesse ueberschritten wird
    } catch (RasterCellOccupiedException e){
      //Fehlermeldungen wenn die Rasterzelle besetzt ist
    }

    return new StdForm(); //Die leere Form wird zurueckgegeben
   }
}
```

Löschen von `ChannelObserver`-Objekten

Mit dem folgenden Befehl kann man von einem π-Programm aus ein `ChannelObserver`-Objekt löschen:

```
channel_name.dispose(<>)
```

wobei kein Argument mitgegeben wird. Durch den obigen Aufruf wird die `invoke(Form f)`-Methode der Klasse `DisposeService` ausgeführt:

```
public class DisposeService extends AbstractService {

  private DrawingView fView_;
  private ChannelObserver chfig_;
  private RasterCell rc_;

...

  protected Form invoke(Form f) {
    //Loesche die Anzeige des ChannelObserver-Objektes aus dem Anzeigefenster
    ChannelObserver fig = (ChannelObserver) fView_.remove(chfig_);

    //Setze die Rasterzelle, in der sich der Kanal befand, als frei
    rc_.setCell_notOccupied();

    return new StdForm();  //Die leere Form wird zurueckgegeben
  }
}
```

Wichtig hierbei ist, dass nur die Anzeige des Kanals aber nicht das Objekt selbst gelöscht wird. Somit kann man weiterhin im Hintergrund mit dem ensprechenden Kanal arbeiten.

Kapitel 7

Anwendung von VisualPi

Das VisualPi Werkzeug wird eingesetzt, um π-Programme zu visualisieren. Es werden die Aktivität der Kanäle und die Zustandsänderungen eines π-Programmes angezeigt. Nebenläufige Agenten führen zu unterschiedlichen Programmzuständen, die mitangezeigt werden. In diesem Kapitel wird die Anwendung von VisualPi anhand von drei Beispielen vorgestellt. Alle Beispiele verwenden die im Abschnitt 2.1 und Kapitel 6 vorgestellten Befehle.

7.1 Beispiel 1

Das erste Beispiel, ein einfaches π-Programm, ist in der Abbildung 7.1, im deren rechten Fenster, dargestellt. Es dient zur Betonung der Eigenschaft, dass π-Programme nicht-deterministisch sind. In diesem Fall wird der Programmablauf nicht immer dieselben Zustände enthalten und die Endergebnisse sind verschieden. Durch Klicken auf den ”execute”-Knopf wird das VisualPi-Anzeigefenster und der Anfangszustand des π-Programmes angezeigt (Abbildung 7.1, linkes Fenster).

Der Anfangszustand des π-Programmes ist gegeben durch vier Kanäle, ”channel1”, ”channel2”, ”channel3” und ”channel4”. Der Name von ”channel1”, ”channel3” und ”channel4” blinkt, da alle drei im *noempty*-Zustand sind, und mindestens eine *get*-Operation wartet darauf, ausgeführt zu werden. Das π-Programm kann weiter durch Aktivierung von ”Step by Step Execution Tool” ausgeführt werden. Der Programmablauf kann unterschiedlich sein, je nachdem welcher Kanal betätigt wird. Es werden weiter unten nur zwei von den möglichen Ablaufszenarios angezeigt.

7.1.1 Szenario 1

Die Programmzustände des ersten Szenarios werden in der Abbildung 7.2 angezeigt. Klickt man im linken Fenster der Abbildung 7.1 auf das ”channel1”-Objekt, so wird die *get*-Operation

```
channel1 ? X > channel3.dispose(<>)
```

ausgeführt und als Folge wird das ”channel3”-Objekt gelöscht. Das Programm wechselt in den Zwischenzustand, der im linken Fenster der Abbildung 7.2 angezeigt wird. In diesem Zustand blinken ”channel1” und ”channel4” weiter. Klickt man noch einmal auf das ”channel1”-Objekt, so wird das ”channel5”-Objekt erzeugt als Folge der Ausführung folgender *get*-Operation:

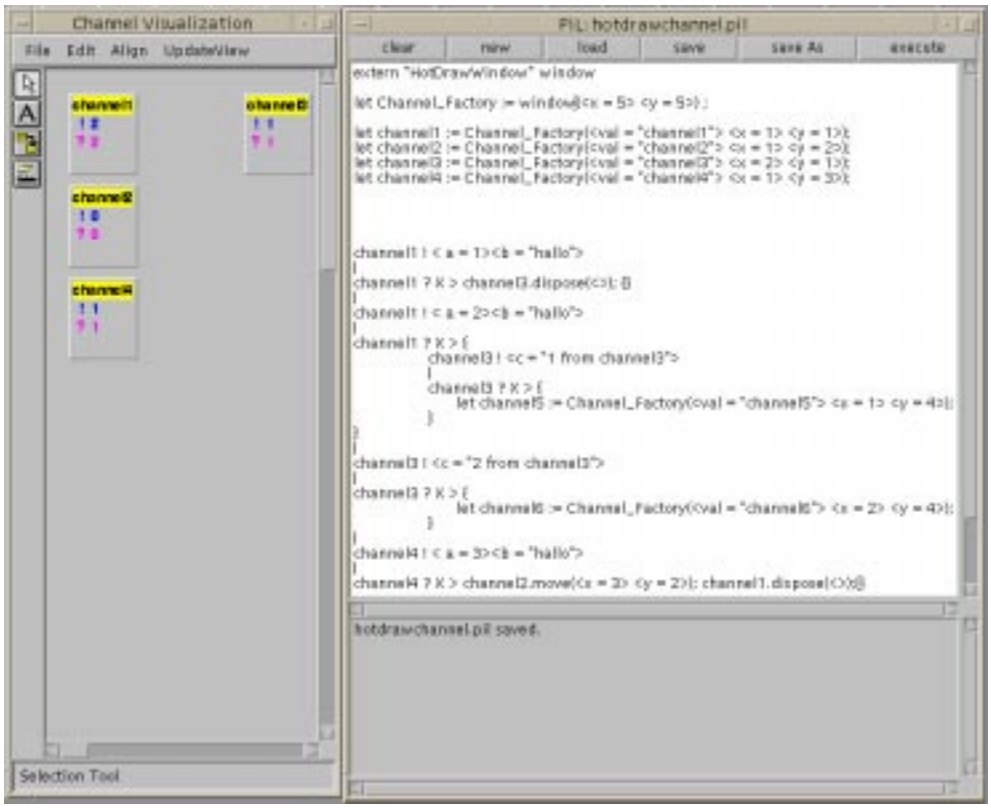

Abbildung 7.1: Beispiel 1. Rechts: das π-Programm, links: Anfangszustand des π-Programmes

```
channel1 ? X > {
    channel3 ! <c = "1 from channel3">
    |
    channel3 ? X > {
         let channel5 := Channel_Factory(<val = "channel5"> <x = 1> <y = 4>);
    }
```

Ein neuer Programmzustand entsteht, der im rechten Fenster derselben Abbildung dargestellt ist. Der einzige Kanal, der in diesem Zustand noch blinkt, ist ”channel4”. Klickt man darauf, so wird die *get*-Operation

```
channel4 ? X > channel2.move(<x = 3> <y = 2>); channel1.dispose(<>)
```

ausgeführt, indem das ”channel1”-Objekt gelöscht und ”channel2” verschoben wird. Somit erreicht das π-Programm sein Endzustand, der im unteren Fenster derselben Abbildung angezeigt ist.

7.1.2 Szenario 2

Dieses Szenario wird in der Abbildung 7.3 angezeigt. Klickt man im linken Fenster der Abbildung 7.1 auf das ”channel3”-Objekt, so wird die *get*-Operation

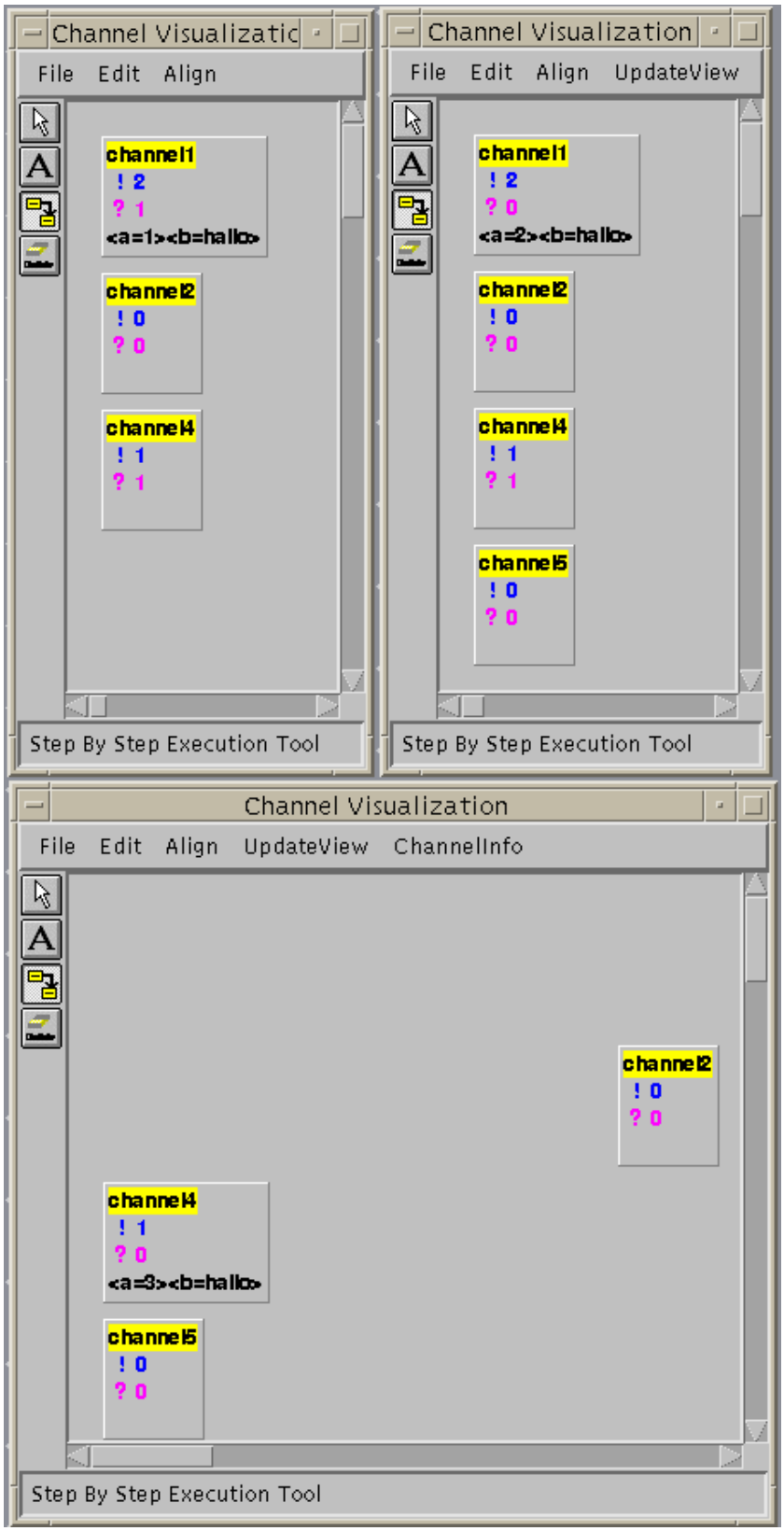

Abbildung 7.2: Beispiel 1: Szenario 1

```
channel3 ? X > {
      let channel6 := Channel_Factory(<val = "channel6"> <x = 2> <y = 4>);
}
```

ausgeführt, indem das ”channel6”-Objekt erzeugt wird. Im neuen Zustand blinken ”channel1”und ”channel4” weiter (linkes Fenster der Abbildung 7.3). Durch Klicken auf das ”channel4”-Objekt wird folgende *get*-Operation ausgeführt:

```
channel4 ? X > channel2.move(<x = 3> <y = 2>); channel1.dispose(<>),
```

indem ”channel1” gelöscht und ”channel2” verschoben wird. Der Endzustand ist in diesem Fall im rechten Fenster der Abbildung 7.3 dargestellt.

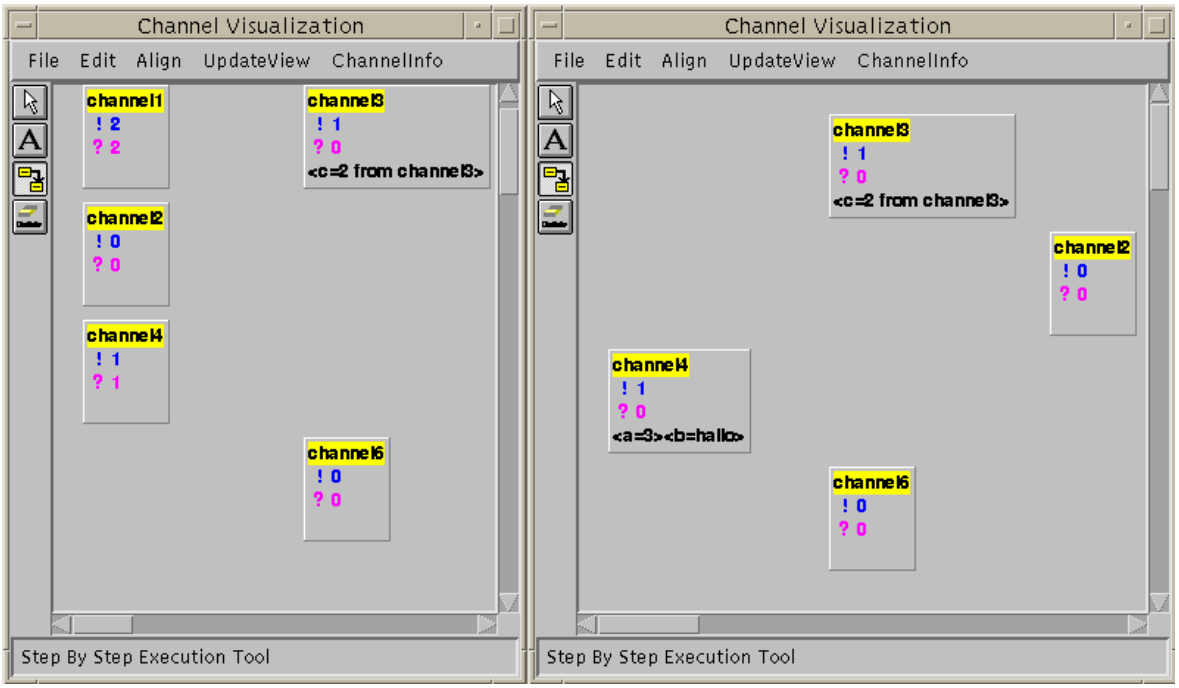

Abbildung 7.3: Beispiel 1: Szenario 2

Der Vergleich der zwei Szenarios zeigt folgende Merkmale, die auf Nicht-Determinismus hinweisen:

- Der Programmablauf enthält im ersten Szenario drei und im zweiten Szenario zwei Zustände.
- Die Zustände aus dem ersten Szenario sind unterschiedlich von denjenigen aus dem zweiten Szenario.
- Die Endzustände sind verschieden.

7.2 Beispiel 2

Dieses Beispiel dient zur Visualisierung eines π-Programmes, das sich mit boolschen Ausdrücken beschäftigt. Das ursprüngliche Programm sieht wie folgt aus:

```
//
// file: bool.pil
//
// shows how boolean can be encoded in pil
// booleans are encoded as forms:

// needed for output:
extern "pi.extern.StdOut" pr

let True := \b > b ? X > X.true ! <>;
let False := \b > b ? X > X.false ! <>;

// test the True
run {
  new truecase
  new falsecase
  run { truecase ?* _ > pr("it's true"); {} }
  run { falsecase ?* _ > pr("it's false"); {} }

  // create the boolean channel
  new b
  b ! <true = truecase><false = falsecase>

  // instantiate True with b
| True ! b
}

let Not := \a > a.out ? X > a.in ! X<true = X.false><false = X.true>;

// test Not True
run {
  new truecase
  new falsecase
  run { truecase ?* _ > pr("it's true"); {} }
  run { falsecase ?* _ > pr("it's false"); {} }

  // create the boolean channel
  new b2
  new c2

  b2 ! <true = truecase><false = falsecase>
| Not ! <in = c2><out = b2>
| True ! c2
}

let Unknown := \b > b ? X > X.unknown ! <>;

// text Not Unknown
run {
  new truecase
  new falsecase
  new unknowncase
  run { truecase ?* _ > pr("it's still true"); {} }
  run { falsecase ?* _ > pr("it's still false"); {} }
  run { unknowncase ?* _ > pr("it's unknown"); {} }
```

```
  // create the boolean channel
  new b2
  new c2

  b2 ! <true = truecase><false = falsecase><unknown = unknowncase>
| Not ! <in = c2><out = b2>
| Unknown ! c2
}
```

Ohne den Einsatz von VisualPi wurde nach der Ausführung des obigen π-Programmes nur das Endergebnis im unteren Fenster der Abbildung 7.4 ausgegeben. Um VisualPi einsetzen zu können, sind im

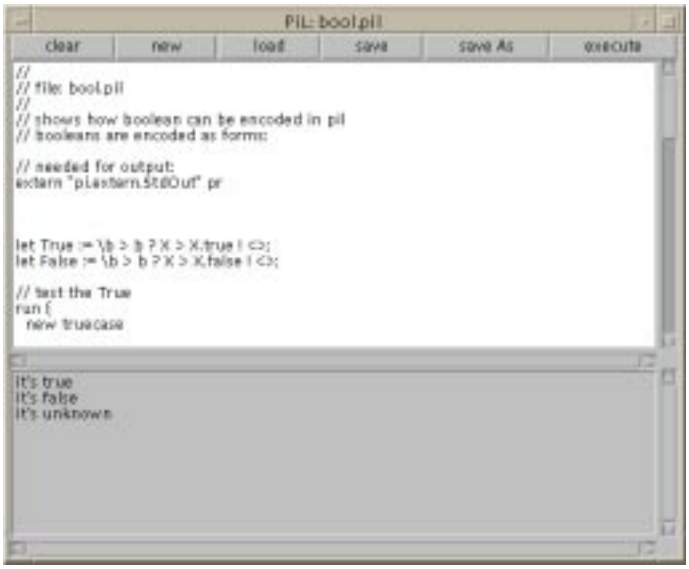

Abbildung 7.4: Beispiel 2

obigen π-Programm nur wenige Änderungen vorzunehmen, die schon im Kapitel 6 beschrieben wurden. Das geänderte π-Programm sieht wie folgt aus:

```
// Shows how boolean can be encoded in pil booleans are encoded as forms

extern "HotDrawWindow" window

// Needed for output
extern "pi.extern.StdOut" pr


//Create ChannelFactory
let ChannelFactory := window(<x = 5> <y = 6>) ;

//create the True channel
let True := ChannelFactory(<val = "True"> <x = 2> <y = 1>);
run{
let myTrue := \b > b ? X > X.true ! <>;
True ?* X > myTrue ! X
}

// Test the True
run {
  let truecase := ChannelFactory(<val = "truecase_true"> <x = 1> <y = 2>);
```

```
  let falsecase := ChannelFactory(<val = "falsecase_true"> <x = 2> <y = 2>);

 run { truecase ? _ > pr("it's true");{} }
  run { falsecase ? _ > pr("it's false"); {} }

// Create the boolean channel
let b := ChannelFactory(<val = "b_true"> <x = 3> <y = 2>);
 b ! <true = truecase><false = falsecase>
|

// Instantiate True with b
True ! b
}

//Create the Not channel
let Not := ChannelFactory(<val = "Not"> <x = 2> <y = 3>);
run{
let myNot := \a > a.out ? X > a.in ! X<true = X.false><false = X.true>;
Not ?* X > myNot ! X
}

// Test Not True
run {

  let truecase2 := ChannelFactory(<val = "truecase_not"> <x = 1> <y = 4>);
  let falsecase2 := ChannelFactory(<val = "falsecase_not"> <x = 2> <y = 4>);

  run { truecase2 ?* _ > pr("it's true"); {} }
  run { falsecase2 ?* _ > pr("it's false"); {} }

  // create the boolean channels
  let b2 :=  ChannelFactory(<val = "b_not"> <x = 3> <y = 4>);
  let c2 :=   ChannelFactory(<val = "c_not"> <x = 4> <y = 4>);

  b2 ! <true = truecase2><false = falsecase2>
| Not ! <in = c2><out = b2>
| True ! c2
}

//Create the Unknown channel
let Unknown := ChannelFactory(<val = "Unknown"> <x = 2> <y = 5>);
run{
let myUnknown := \b > b ? X > X.unknown ! <>;
Unknown ? X > myUnknown ! X
}

// Test Not Unknown
run {

  let truecase := ChannelFactory(<val = "truecase_un"> <x = 1> <y = 6>);
  let falsecase := ChannelFactory(<val = "falsecase_un"> <x = 2> <y = 6>);
  let unknowncase := ChannelFactory(<val = "unknowncase"> <x = 3> <y = 6>);

  run { truecase ?* _ > pr("it's still true"); {} }
  run { falsecase ?* _ > pr("it's still false"); {} }
  run { unknowncase ?* _ > pr("it's unknown"); {} }

  // create the boolean channel
  let b2 :=  ChannelFactory(<val = "b_unknown"> <x = 4> <y = 6>);
  let c2 :=   ChannelFactory(<val = "c_unknown"> <x = 5> <y = 6>);

  b2 ! <true = truecase><false = falsecase><unknown = unknowncase>
```

```
| Not ! <in = c2><out = b2>
| Unknown ! c2
}
```

Bei der Ausführung des obigen π-Programmes wird von VisualPi das Fenster aus der Abbildung 7.5 erzeugt. Alle Befehle der Form

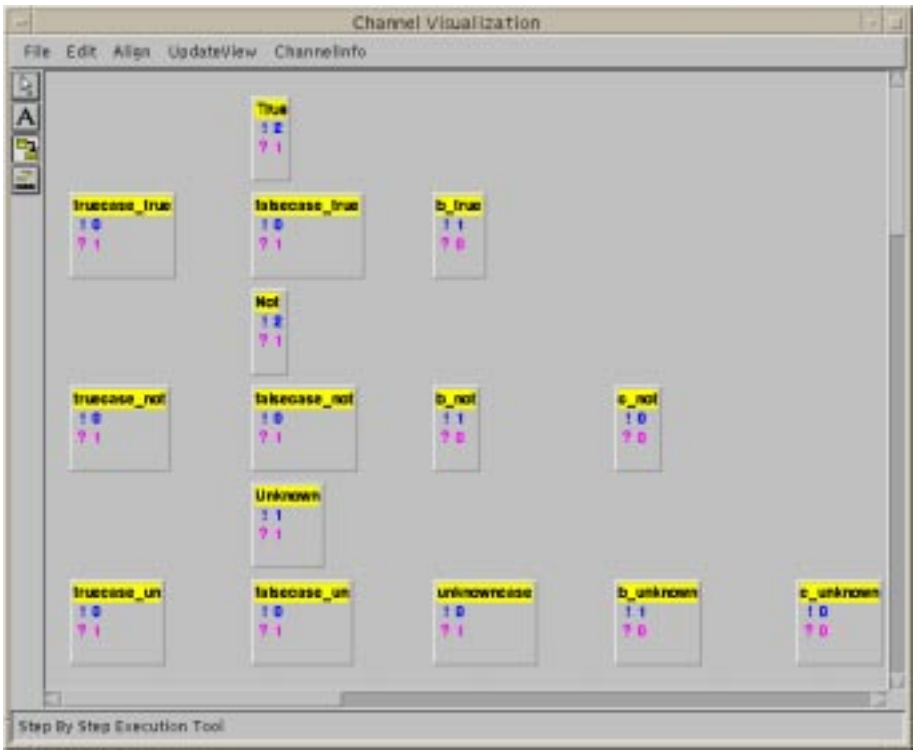

Abbildung 7.5: Beispiel 2: VisualPi-Anzeigefenster

```
let channel_name := Channel_Factory(<val ="channel_name"><x = Val1><y = Val2>);
```

erzeugen ein Kanal. Im VisualPi-Anzeigefenster werden die Kanäle angezeigt, die während des π-Programmes erzeugt werden. Die Objekte ”True”, ”Not” und ”Unknown” blinken im angezeigten Anfangszustand des π-Programmes. Somit kann der Benutzer durch Aktivierung des ”Step by Step Execution Tool” und Klicken eines der drei blinkenden Objekte die Ausführung des π-Programmes fortsetzen. Durch schrittweises Klicken kommt man zum selben Endergebnis (Abbildung 7.4, unteres Fenster), mit dem Unterschied, dass der Benutzer alle Zwischenzustände des π-Programmes mitverfolgen kann.

7.3 Beispiel 3

In diesem Beispiel handelt es sich um ein π-Programm, das eine nebenläufige Warteschlange simuliert. Der Code und das Ergebnis des ursprünglichen π-Programmes sind der Abbildung 7.6 zu entnehmen.

Dieses Beispiel unterstreicht die Nebenläufigkeit, die π-Programme charakterisiert. Der Code dieses Beispiels wird ebenfalls angepasst, um den Programmablauf mit VisualPi anzeigen zu können:

```
// A concurrent queue
// F. Achermann
// Modified by F. Achermann and C. Cris

extern "pi.extern.StdOut" pr
extern "pi.extern.Concat" ++
extern "pi.extern.Addition" +

extern "visualPi.HotDrawWindow" window

//Create ChannelFactory
let ChannelFactory := window(<x = 6> <y = 7>) ;

//Create the get channel
let get := ChannelFactory(<val = "get"> <x = 3> <y = 1>);


//Queue Initializations
let link := ChannelFactory(<val = "link"> <x = 1> <y = 2>);
let init := ChannelFactory(<val = "init"> <x = 3> <y = 2>);
run {link ! init}
run {init ! <>}

// The head acceps a 'get' request to yield its
// value and trigger the next cell.
let head := ChannelFactory(<val = "head"> <x = 1> <y = 4>);
run {
        head ?* x > { init.dispose(<>) ;
              {get ? y > { y ! <val = x><result = x.result> | x.next ! <> } } }
    }

// A cell waits to be triggered (something read from along x.ready)
// and then itself becomes the head of the queue
let cell := ChannelFactory(<val = "cell"> <x = 1> <y = 3>);
run {
cell ?* x > x.ready ? _ > head ! <val = x><next = x.next><result = x.result>
      }

//Definition of the counter (see text)
let newCounter := \x. (
    new counter
    run counter ! x
    def getNext y > {
        counter ? pos > { y.reply ! pos | counter ! (pos + 1) }
    };
   getNext
);

//Initializations of the counter for "next" and "result" channels
let nextTitelCounter := newCounter(1);
let resultTitelCounter := newCounter(1);
let nextPosCounter := newCounter(1);
let resultPosCounter := newCounter(1);

//Create the put channel
let put := ChannelFactory(<val = "put"> <x = 1> <y = 1>);
//Definition of the put operation on the queue
run {
         put ?*  x > {
            link ? ready > {
               let next := ChannelFactory(<val = "next " ++ (nextTitelCounter(<>))>
                                          <x = (nextPosCounter(<>))> <y = 6>);
```

```
                let result := ChannelFactory(<val = "result "++ (resultTitelCounter(<>))>
                                             <x = (resultPosCounter(<>))> <y = 7>);
                cell ! x<ready = ready><next = next><result = result>
                | result ! <val = x.val>
                | link ! next
                | x.reply ! <>
            }
        }
    }

//Main run
run {
let r := ChannelFactory(<val = "r"> <x = 1> <y = 5>);
  put("one") ;put("good"); put("turn"); put("deserves"); put("another");{}
  | get ! r | get ! r | get ! r | get ! r | get ! r
  | r ?* x > {x.result ? x > pr(x.val); {}}
}
```

In der simulierten Warteschlange werden nacheinander fünf Strings gelegt: ”one”, ”good”, ”turn”, ”deserves”, ”another”. Bei jedem Speichern eines Strings werden ein ”next” und ein ”result” Kanal kreiert. Damit man die erzeugten ”next”- und ”result”-Objekte im VisualPi-Anzeigefenster während des Programmablaufes voneinander unterscheiden kann, wurden sie, durch Einsetzen eines zusätzlichen *counter*, durchnummeriert. Die Erzeugung und Anzeige jedes ”result”-Objektes entspricht jeder Ausführung einer *put*-Operation und dadurch jeder Ablegung eines Strings in die Warteschlange. Erst wenn ein ”result”-Objekt blinkt und darauf mit der Maus geklickt wird (”Step by Step Execution Tool” muss aktiv sein), wird der entsprechende String aus der Warteschlange geholt und als Ergebnis ausgegeben. Auch hier wird der Ablauf des π-Programmes vom Benutzer durch Klicken auf die verschiedenen Kanäle kontroliert und verfolgt. In den Abbildungen 7.7 und 7.8 ist der Anfangszustand bzw. ein Zwischenzustand des π-Programmes aufgenommen worden.Im Anfangszustand blinkt nur das ”put”-Objekt und der Zwischenzustand entspricht dem Programmzustand, in dem gerade auf das ”result1”-Objekt geklickt und den String ”one” als Ergebnis ausgegeben wurde. Die zwei Aufnahmen dienen nur zur Illustration und können auf gar keinen Fall das Vergnügen ersetzen, das bei der interaktiven Ausführung des π-Programmes entsteht.

Abbildung 7.6: Beispiel 3: Nebenläufige Warteschlange

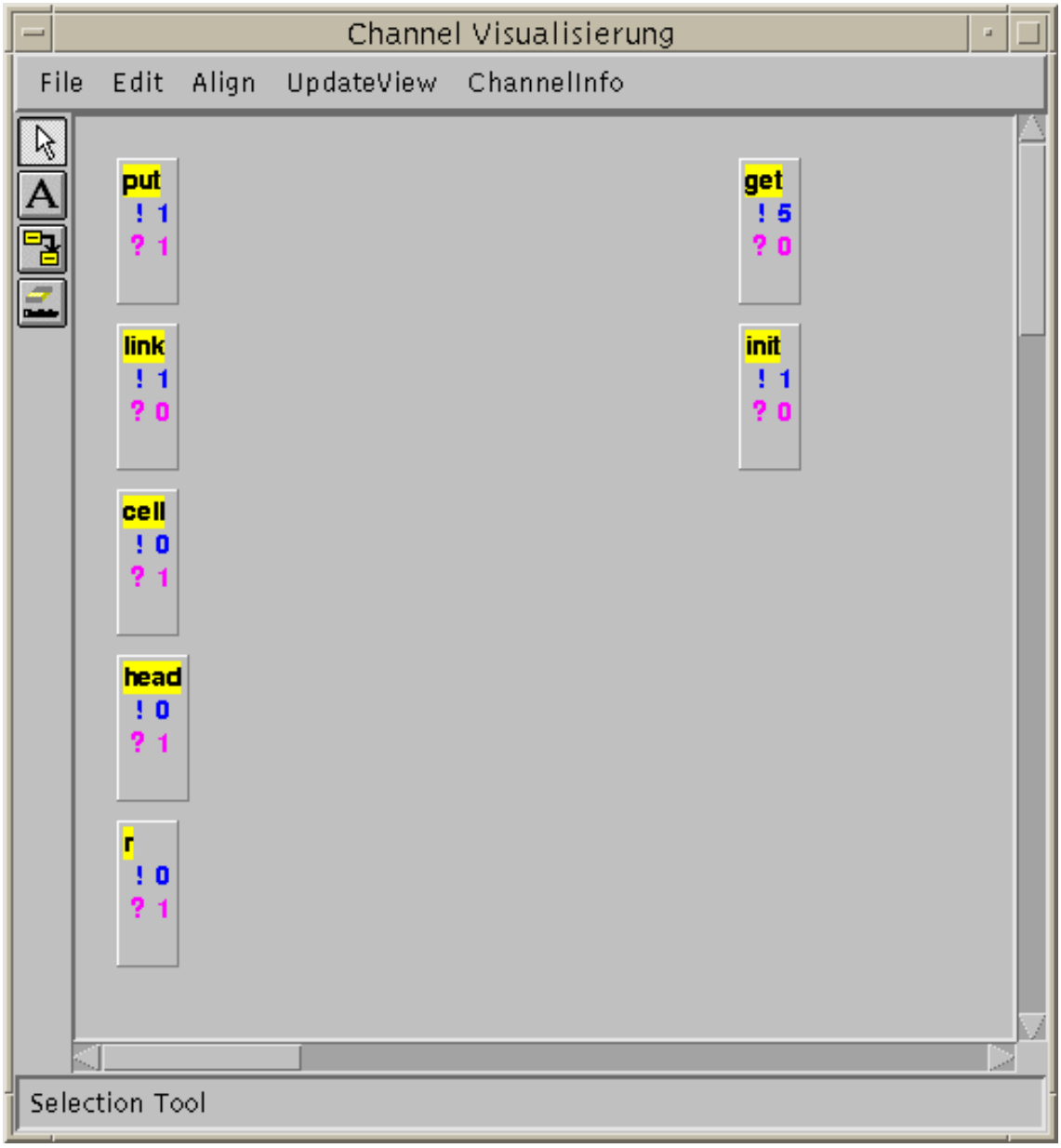

Abbildung 7.7: Beispiel 3: Anfangszustand des π-Programmes

Abbildung 7.8: Beispiel 3: Zwischenzustand des π-Programmes

Kapitel 8

Schlussfolgerungen

In diesem Kapitel werden die Arbeit kurz zusammengefasst und einige Erfahrungen und daraus abgeleitete Empfehlungen weitergegeben. Abgeschlossen wird das Kapitel mit einem Ausblick auf mögliche Erweiterungen des VisualPi Werkzeuges.

8.1 Zusammenfassung der Arbeit

In den Kapiteln 2 und 3 wurden die grundlegenden Konzepte der JPict und JHotDraw Frameworks vorgestellt. Dabei wurden auf der einen Seite Begriffe wie ”Kanal” und ”Form” erläutert und auf der anderen Seite die Komponenten vorgestellt, die an einem Grafikeditor teilnehmen. Die Hauptkapitel der vorliegenden Arbeit sind 4 (Desing von VisualPi) und 5 (Implementation von VisualPi). Im Kapitel 4 wurde das Design mit Hilfe von Klassendiagrammen und danach im Kapitel 5 die Implementation und Einbindung in PiL von VisualPi beschrieben. Drei Beispiele wurden im Kapitel 6 vorgeführt, mit Zustandsaufnahmen der entsprechenden π-Programme.

8.2 Rückblick

VisualPi baut auf dem JHotDraw Framework auf, um den Ablauf eines π-Programmes zu visualisieren. Dafür wurde mit zwei separaten, komplexen Frameworks JPict und JHotDraw gearbeitet. Die Einarbeitung in die beiden Frameworks nimmt eine ganz andere Arbeitstechnik in Anspruch als die Entwicklung einer kleinen, aus wenigen Codezeilen bestehenden Applikation. Damit später von meinen Erfahrungen profitiert werden kann, sollen hier die wichtigsten Aspekte kurz aufgezeigt werden.

8.2.1 Erfahrungen mit dem JPict Framework

Obwohl JPict ein komplexes Framework ist, war der Aufwand zur Einbindung des VisualPi Werkzeuges in JPict gering. Das Framework hat sehr gut definierte Schnittstellen, so dass mittels Verwendung weniger Klassen oder *interfaces* die Kommunikation zwischen VisualPi und JPict ermöglicht wurde. Der Einarbeitungsaufwand in JPict war, dank der Hilfe von Franz Achermann, nur gering. Ich möchte mich an dieser Stelle bei Franz Achermann für die Betreuung dieser Arbeit bedanken.

8.2.2 Erfahrungen mit dem JHotDraw Framework

Das JHotDraw Framework ist ein auf *Design Patterns* basierender Rahmen. Da es in Java entwickelt wurde, konnte man mittels *interface*-Definitionen klare Trennung zwischen der Design- und Implementationsebene schaffen. Dies vereinfachte das Verständnis des Frameworks und ermöglichte problemlose Erweiterungen. Leider stand, ausser dem gewöhnlichen Codekommentar und einigen Präsentationsfolien, sehr wenig Dokumentation über das Framework zur Verfügung. Dies verlängerte nur die Einarbeitungszeit (das Framework enthält ca. 140 Klassen), die benötigt wurde. Anderseits ermöglichte der gute Aufbau des Frameworks die Einführung vieler Erweiterungen und Anpassungen in VisualPi.

8.3 Ausblick

Die vorliegende Arbeit stellt einen Versuch dar, π-Programme zu visualisieren. Sie erschöpft auf gar keinen Fall alle Visualisierungsmöglichkeiten, die mittels JHotDraw zustande gebracht werden können. In PiL können sehr komplexe Programme geschrieben werden. Die Programme bestehen nicht nur aus Kanälen, die mit ”new” (Abschnitt 2.1) definiert wurden, sondern komplexere Strukturen und Funktionen (Beispiele 7.2 und 7.3) können ebenfalls eingeführt werden. Im aktuellen Zustand von VisualPi können nur die einfachen Kanäle visualisiert werden. Eine mögliche Erweiterung ist die Visualisierung von Funktionen und komplexeren Strukturen. Eine andere Möglichkeit kann dadurch entstehen, dass ein anderes Grafikframework als JHotDraw verwendet wird. Das *Actor Event* Modell wurde für VisualPi eingesetzt (Unterkapitel 4.2.3). Dadurch ist jedes `ChannelObserver`-Objekt der *Observer* eines Kanals. In einem nächsten Schritt könnte man das etwas komplexere *Causal Interaction* Modell einführen, indem man mehrere `ChannelObserver`-Objekte in eine Visualisierungsgruppe zusammenfügt und von einem *Coordinator* beobachtet werden (Abschnitt 4.2.2). Dadurch kann man für komplexere π-Programme eine bessere Übersicht des Programmablaufs erlangen.
Visualisierung von Programmen ist ein relativ neues Gebiet, das in der Zukunft an Bedeutung gewinnen wird, und ein Grund dafür ist die Komplexität der Programmen, die ständig steigt.

Literaturverzeichnis

[1] Robin Milner. The polyadic pi calculus: a tutorial. Ecs-lfcs-91-180, Computer Science Dept., University of Edinburgh, October 1991.

[2] Franz Achermann. JPict - a framework for pi agents. Technical report, October 1998.

[3] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27 of *10*, pages 63–76, October 1992.

[4] Kent Beck und Ralph Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, LNCS 821, pages 139–149. Springer-Verlag, July 1994.

[5] Erich Gamma. JHotDraw5.1. http://members.pingnet.ch/gamma/, March 1998.

[6] Grandy Booch James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, October 1998.

[7] Franz Achermann. An interpreter for PiL. Unpublished, January 1998.

[8] Mark Asley und Gul A. Agha. A visualization model for concurrent systems. In *2nd Joint Conference on Information Sciences*, December 1995.

[9] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[10] Erich Gamma et al. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.