NAIST-IS-DT9961002

博士論文

形式的手法を用いたインタラクティブシステム設計法

池田 瑞穂

2002 年 3 月 12 日

奈良先端科学技術大学院大学
情報科学研究科 情報処理学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

池田 瑞穂

審査委員：　関 浩之 教授
渡邉 勝正 教授
松本 健一 教授

形式的手法を用いたインタラクティブシステム設計法*

池田 瑞穂

内容梗概

アプリケーション開発工程において，ユーザインタフェース（UI）部分の開発の占める割合が大きくなっている．その信頼性の向上はもとより，コスト削減，開発時間の短縮が要求されている．UI 設計は，従来のソフトウェア設計と異なり，設計対象の振る舞いや入出力関係に関する設計（機能設計）に加えて，画面上のレイアウトなどプレゼンテーションレベルでの詳細設計（プレゼンテーション設計）が必要である．しかし，視覚面に重点を置いた設計となりがちであり，機能設計とプレゼンテーション設計間の整合性が取れないことが多い．

対象システムのユーザの作業手順を分析・モデル化する様々な設計法や設計技術が提案されている．しかし，これらの設計手法は，機能設計あるいはプレゼンテーション設計の個別に目的を絞ったものであり，相互の関連や影響を考慮した設計方法論は少ない．これらの関連を考慮した，体系化されたインタラクティブシステム設計法が望まれる．

本研究では，設計の初期段階から適用できる，機能設計とプレゼンテーション設計の一貫性を考慮した，インタラクティブシステム設計法を提案する．まず，タスク図という図的言語を用いてタスクモデルを記述する手法を提案する．そして，タスク図の意味を，プロセス代数に基づく記述法である LOTOS を用いて形式的に定義する．次に，LOTOS 仕様に対するモデル検査法を用いて，タスク図で記述されたタスクモデルを形式的に検証する方法を述べる．また，タスクモデルから HTML，CGI を実装アーキテクチャとしたプロトタイプを自動生成する方法を説明する．本論文では，書籍の販売システムを例題として本設計法の説明を行う．この例題に対して形式的検証，プロトタイプ自動生成を行った結果について述べ，その有効性について議論する．

次に，代数的仕様記述の部分クラスである抽象的順序機械型仕様 (ASM 仕様) を用いてユーザインタフェースの仕様を形式的に記述し，その記述に従ってプロトタイプ自動

*奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 博士論文, NAIST-IS-DT9961002, 2002 年 3 月 12 日.

生成を行う手法を提案する．タスクモデルからプロトタイプを作成する際，代数的仕様記述を行うことにより，より厳密に UI 設計情報を取り扱うことができる．まず，個々の UI モジュールがメッセージ送受信によって非同期に動作するような，簡潔な多プロセスモデルである抽象的ウィンドウシステム（AWS）モデルを導入する．そして，AWS に基づいて UI の ASM 仕様を記述する方法について述べる．また，Java を実装アーキテクチャと仮定し，UI の ASM 仕様からプロトタイプを自動生成するための枠組を提案する．この枠組に従って ASM 仕様を Java プログラムに変換するコンパイラを作成した．本論文では，コンパイラを用いて仕様記述例をコンパイルした結果についても述べる．

キーワード

インタラクティブシステム，ユーザインタフェース，形式的記述，形式的検証，タスク図，代数的仕様記述，抽象的順序機械，コンパイラ

A Formal Method in Interactive System Design*

Mizuho Ikeda

Abstract

In development of an application system, designing the user interface (UI) comes to occupy a larger part. Not only the improvement of the reliability but also shortening the cost and the development term are demanded. UI design needs a detailed design in the presentation level such as the layout of widgets on screens (the presentation design) as well as the functional design which guarantees the correct input/output behavior of the target system. However, designers tend to pay too much attention to the presentation design. As a result, the designed system often does not satisfy the consistency between the functional design and the presentation design.

Various techniques have been proposed and used for interactive system design. However, these techniques focus on only a part of the presentation design or the functional design. Also, it is often pointed out that a design process of an interactive system highly depends on designers' experience and lacks a systematic methodology.

In this thesis, I propose a formal method for the interactive system design. In the proposed method, a task model is constructed by using task flow diagram. The semantics of the task flow diagram is formally defined by LOTOS, which is a formal specification language based on the process algebra. A task flow diagram is refined by decomposing each task into several subtasks which represent system actions and user actions. Next, I present a formal verification method for a task model by using a model checking tool and also present an automatic prototype generation method which translates a task model into an HTML and a CGI program. The proposed method is explained by using on-line book purchasing problem.

*Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9961002, March 12, 2002.

Next, I propose a formal description method of user interface (UI) based on a subclass of algebraic specifications called abstract sequential machine (ASM) specifications and also discuss a prototype generation method from an ASM specification. I introduce an abstract window system (AWS), which is a simple multiprocess model where each UI module behaves asynchronously by sending or receiving messages. Each UI module is modeled as a component of the AWS and is defined by an ASM specification. We implemented a compiler which translates an ASM specification of UI into a Java program. Sample specifications and the compiled programs are presented.

Keywords:

Interactive System, User Interface, Formal Specification, Formal Verification, Task Flow Diagram, Algebraic Specification, Abstract Sequential Machine

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究の背景と目的

ソフトウェアシステムの多機能化，ユーザの多様化により，アプリケーション開発工程においてユーザインタフェース (UI) 部分の開発の重要度が高まっている [1]．従来のソフトウェア設計と異なり，UI 設計では，設計対象の振る舞いや入出力関係に関する設計（機能設計）に加えて，画面上のレイアウトなどプレゼンテーションレベルでの詳細設計（プレゼンテーション設計）が必要である．現在 Visual Basic[2] や Delphi[3]，JBuilder などのグラフィカルユーザインタフェース (GUI) 構築支援ツールが充実してきており，GUI を作成すること自体は容易になっている．そのため開発者は，画面の構成，ボタンの配置や表示される文言など，各画面ごとの視覚的な部分を重視した，すなわちプレゼンテーション設計に比重を置いたプロトタイプを作成しがちになっている．このように視覚的な部分を重視した開発法では，(1) 自然な業務手順・作業手順が成果物に十分反映されない，(2) 作業手順が作業間の依存関係に矛盾していても見逃しやすい，といった問題が生じている．

インタラクティブシステムを構成するソフトウェアの構造は，一般に，プレゼンテーション設計の実装にあたるプレゼンテーション層，ユーザの振る舞いを表現しその実現を行なうビジネスロジック層，データベース部分であるデータ層からなると考えられる．GUI の設計は，このソフトウェアの 3 層構造におけるプレゼンテーション層の設計に相当する（図 1.1）．

対象システムのユーザの作業手順を分析・モデル化する手法として，ワークフロー分析・オブジェクト指向分析等が，また，モデル化のための言語として UML(Unified Modeling Language)[4] 等が提供されている．しかしこれらの手法は，ソフトウェアの 3 層構造の各層に個別に目的を絞ったものであり，相互の関連や影響を考慮した設計方法論は少ない．また，それらを用いて構築した設計を評価する手法は少ない．

また UI 設計では，上流工程において最終成果物のユーザビリティ等を正確に予想する

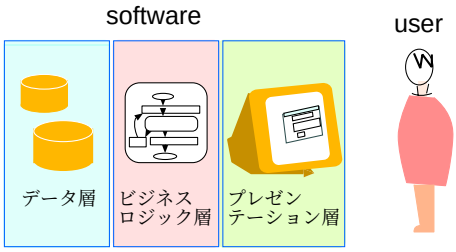


図 1.1 ソフトウェアの 3 層構造モデル

ことは困難である．プロトタイプの評価・修正を繰り返し，品質を逐次改善していく必要があり，そのためには仕様からのプロトタイプ自動生成が可能であることが望ましい．

これらの問題を解決するためには，自然でかつ作業間の依存関係に矛盾しない業務手順の実現を重視した設計法を採る必要がある．形式的に意味定義された仕様記述法を採用することにより，仕様の厳密な記述，ならびに初期設計から実装段階への不整合のない段階的詳細化が行える．また，形式的検証により仕様の正当性を保証することやプロトタイプを自動生成することも可能となる．

本論文では，設計の初期段階から適用できる，機能設計とプレゼンテーション設計の一貫性を考慮した，インタラクティブシステム設計法（**タスクモデルを用いた手法**）を提案する．この設計法ではまず，タスク図という図的言語を用いてタスクモデルを記述する．タスク図の意味は LOTOS を用いて形式的に定義されている．そして，タスクモデルを形式的に検証し，プロトタイプを自動生成する．

また，代数的仕様記述の部分クラスである抽象的順序機械型仕様 (ASM 仕様) を用いて UI の仕様を形式的に記述し，その記述に従ってプロトタイプ自動生成を行う手法（**代数的仕様記述を用いた UI 設計法**）の提案も行う．まず，個々の UI モジュールがメッセージ送受信によって非同期に動作するような，簡潔な多プロセスモデルである抽象的ウィンドウシステム（AWS）モデルを導入する．この設計法では AWS モデルに基づいて UI の ASM 仕様を記述する．UI の ASM 仕様からプロトタイプを自動生成するための枠組を提案し，これにしたがって ASM 仕様を Java プログラムに変換するコンパイラを作成した．タスクモデルを用いた手法では，UI 設計情報の形式的意味を与えられていなかったが，代数的仕様記述を用いた UI 設計法と組み合わせることで，UI 部品の挙動や画面切替などまで含めた，システムの詳細な挙動を厳密に取り扱えるようになる．ただし，本論文ではタスクモデルを用いた手法とは独立した設計法として扱う．

以降，本章では両提案手法の概略を説明し，関連研究について述べる．

1.2　タスクモデルを用いたインタラクティブシステム設計法

1.1 節で述べたように，本設計法では，ソフトウェアの 3 層構造モデル（図 1.1）の各層の関連や影響を考慮し，かつ仕様の厳密な記述が可能な，インタラクティブシステムのための仕様記述法が望まれる．ここでは，ユーザタスク（例えば，通信販売で本を購入する）の構造の形式的仕様記述（**タスクモデル**）に基づく設計法を提案する．そしてその形式的仕様に対して，形式的検証やプロトタイプ自動生成を行ない，それらを繰り返すことにより，品質の高いプロトタイプが得られる．

本設計法ではタスクモデルをシステムの主たる仕様として扱う．タスクモデルとは，タスクの構造を適切な記述法を用いて書き下したものである．タスクの構造とは，そのタスクがどのようなサブタスクをどのような順序で行うことで実現されるかを表すものである．本設計法ではタスクモデルの記述法として，**タスク図**という図的言語を用いる．これは UML において定義されているアクティビティ図に基づいた記述法である．タスク図を使うことでタスク構造を図的にわかりやすく表現できる．また，タスク図の意味はプロセス代数に基づく記述法である LOTOS[5, 6] を用いて形式的に定義されているため，タスクモデルの形式的検証やプロトタイプの自動生成が可能となる．

本設計法は以下のような特長を持つ．

(1) **作業手順（ユーザタスク）の記述を設計における主たる要素とする．**
　システムの機能の仕様を重視した設計技法やプレゼンテーション層主導の設計法では，ユーザタスクの構造や流れが設計プロセスに反映されにくかった．そこで，ビジネスロジック層の仕様であるユーザタスクの形式的記述（タスクモデル）を設計法の軸とし，プレゼンテーション層，データ層とのインタフェースを含んだ記述を行なう．これにより，初期段階からユーザタスクの流れを意識した設計が可能となり，最終成果物におけるユーザの作業の生産性の向上を期待できる．

(2) **仕様の形式的記述と詳細化を行う．**
　初期仕様を形式的記述法を用いて記述し，前段階の仕様を満たすように設計のプロセスを進める．これにより設計のプロセス間の不整合を防止できる．また，形式的記述を行うことで，プロトタイプの自動生成や仕様の形式的検証が可能となり，システム設計の効率化を計ることができる．

タスクモデルには，タスク構造に加え，タスク内で扱われるデータに関する記述（どのタスクにおいてどのようなデータがユーザ・システム間でやり取りされるか，また，タ

スク間にどのようなデータ依存関係があるか）も行う．各タスクにおけるこれらの記述をそのタスクの**データ属性**と呼ぶ．タスク図の意味は，イベントにおいて送受信されるデータを抽象データ型に基づいて表現できるフル LOTOS を用いて定義される．

タスク図のアクティビティ図に対する違いは以下のとおりである．

- データ属性の記述ができるように拡張されている．
- 定義を簡潔にするため，並行動作部分は並行タスクという一頂点で表す．
- LOTOS を使って意味が定義されている．

本研究ではまずタスク図の構文と意味を定義した．そして，モデル検査法を使った形式的検証手順を提案し，例題に対して検証を行なった．また，プロトタイプ生成法を考案し，WWW アーキテクチャのためのプロトタイプ生成システムを実装した．これらの詳細については 2 章で述べる．

1.3 代数的仕様記述を用いた UI 設計法

本研究では UI 設計を行なうもう一つの方法として, 代表的な形式的記述法の一つである代数的仕様記述法 [7] を用いた設計法を提案する. この設計法は，(1) 機能設計を論理的不整合なく行なうことを重視し，(2) 具体化レベルの仕様からのプロトタイプ自動生成が可能である点で，本研究で目的とする設計法の一つの実現となっている．しかしタスクモデルを主たる仕様として扱うという考えは陽には含まれていない．代数的仕様記述は，要求定義レベルから具体化レベルまで任意の抽象レベルで記述が行なえるという利点を持つが，特に具体化レベルの代数的仕様記述をタスクモデルを用いた手法と組み合わせることにより，UI 部品の挙動や画面切替などまで含めたシステムの詳細な挙動を厳密に取り扱えるようになる．ただし，本論文ではタスクモデルを用いた手法とは独立した設計法として扱う．

代数的仕様記述法は, 等式論理に基づき仕様の意味が簡明に定義される, 要求定義レベルからプログラムに対応する具体化レベルまで任意の抽象レベルで記述が行なえるなどの利点をもつ. 代数的仕様記述法の部分クラスとして, 抽象的順序機械型代数的仕様 (以下, ASM 仕様と略記) がある [8]. 一つの ASM 仕様は, 状態遷移関数, 状態成分関数の 2 種類の関数の宣言[1]および, 公理の集合からなる. 各公理では, ある状態遷移関数の適用に

[1]一般には, これに加えて補助関数を許す.

より, 各状態成分関数の値がどのように更新されるかを, 遷移前の状態成分関数の値の組合せによって定義する. ASM 仕様は, 代数的仕様記述法一般の利点に加えて, 以下のような特長をもつ.

- 左線形かつ重なりを持たない項書換え系の部分クラスと見なせることから無矛盾性が保証される. 同様に, 完全性の検査も容易である [8].
- 状態遷移関数を手続き, 状態成分関数を変数に対応づけることにより, 効率の良いプログラムへの変換が可能である. ASM 仕様から手続き型プログラムへのコンパイラも試作されている [9].
- 仕様の形式的検証を行なう際に, 状態を表す項に関する構造的帰納法を用いるなど, 検証の方針が立て易い [10].

本研究では, 抽象的ウィンドウシステム (AWS) と呼ばれるモデルを導入する. AWS は, 個々の UI モジュールがメッセージ送受信によって非同期に動作するような簡潔な多プロセスモデルである. AWS に基づき UI の ASM 仕様を記述する. そこでは, 個々の UI モジュールは一つの ASM として定義される. AWS 自身も, 個々の UI モジュール, およびメッセージ処理用キューを状態成分としてもつ一つの ASM として定義される.

次に, Java を実装アーキテクチャとし, UI の ASM 仕様を実装するための枠組を提案する. その枠組にしたがって, ASM 仕様からプロトタイプを自動生成するコンパイラを試作した. 概念的には, UI 部品および設計対象 UI のどちらも, AWS における UI モジュールであり, 代数的仕様においては, それぞれ一つの ASM 仕様によってその振舞いが定義される. 現在設計の対象となっている UI を「設計対象 UI」と呼ぶ. UI 部品と設計対象 UI は, 実装において次のように扱いが異なる.

- 設計対象 UI の ASM 仕様を記述し, それをコンパイラに与えることにより, プロトタイプを生成する.
- UI 部品については, その ASM 仕様と実装の両方をあらかじめライブラリとして与えておく.
  - UI 部品の ASM 仕様は, その部品の利用者 (UI の設計者) へのインタフェースとして利用され, また, 設計対象 UI の ASM 仕様と合わせて, 閉じた ASM 仕様を形成する.

– 一方, UI 部品の実装は, 設計対象 UI の ASM 仕様のコンパイル時にリンクされる.

UI 部品の ASM 仕様においては, プレゼンテーションレベルの詳細すべてが記述されている必要はない.

ASM 仕様ではその特長として, 状態成分関数ごとに独立に公理を記述できる. したがって, 仮にプレゼンテーションレベルの意味を ASM 仕様に追加したい場合でも, 必要な状態成分関数とその意味を定義する公理を追加するだけでよく, 既に記述した部分を変更する必要はない. さらに, 部品ライブラリは固定のものではなく, 必要に応じて随時追加すれば良い. 以上の方針に基づいて ASM 仕様から Java プログラムへのコンパイラを作成した. 従来のコンパイル法 [9] と異なり, 本コンパイル法では, 各 ASM を Java の一つのクラスに対応させることにより, 複数の ASM を含む仕様（ある ASM が，別の ASM の状態成分になっている場合も含む）をコンパイルすることが可能である.

なお, 本論文では, UI の ASM 仕様の形式的検証の問題は取り扱わない. また, 形式的記述法は一般に熟練度が高くない設計者にとって理解しにくいという問題点がある. これについては, 可読性を高めるために, ASM 仕様から状態遷移図と自然語による説明文の自動生成システム [11] が利用できる.

1.4 関連研究

形式的記述法を UI 設計に応用する研究は多くなされている.

GUI の仕様を形式的に与えることで，その形式的検証とプロトタイプ自動生成を行う方法が研究されている [12]. これは，1.2 節の特長 (2) に関して本論文と同じ目的を持つものである. しかし，これはユーザのタスクを考慮したモデルではなく，仕様作成段階の方法論や仕様記述からのプロトタイプ自動生成も提案されていない. ユーザタスクの分析を行い，形式的手法を用いてモデル化する研究には以下のようなものがある. Moher ら [13] は，ユーザおよびシステムのモデルをそれぞれペトリネットで記述し，ユーザがシステムを使用する際に起こる不具合をモデル上で検出する手法を提案している. しかし，ペトリネットはタスク構造やデータに関する記述を自然に書き表すのが難しい. Clark ら [14] の方法は，仕様作成段階で，LOTOS[6]，OMT 記法 [15] など複数の記述法を用いて Agent View と呼ばれるユーザモデルを記述する. Agent View は，ユーザとシステム間

のインタラクションの記述であるが，これにシステム内部の制御情報を加えることによりシステムモデルを得る．この設計法は上流工程の仕様の作成を目指したものであり，プロトタイプ生成には言及していない．また，複数の記述法を用いるため，その間の整合性をとるのが難しい．逆に，UI 部品の組み合わせやイベントの実行タイミングの詳細な設計に注目している研究もある [16, 17, 18]．これらの研究ではいずれも形式的検証については検討されていない．d'Ausbourg ら [17] は，Lustre と呼ばれる時系列記述言語と，TRIO と呼ばれる時相論理を用いて,「ユーザは，画面 x から画面 y に z ステップ以下の操作で切り換えられる」など，ユーザビリティに関する性質のモデル検査法による検証，テスト系列自動生成システムを開発している．ユーザタスク全体の流れよりもユーザの操作列に重点がおかれている点で本論文とは目的が異なるが，ユーザビリティに関する性質の形式的検証が可能であることを実証した点で興味深い．これに対して本研究では，タスク記述には直観的にわかりやすい図的言語を用いることができる一方で，その図的言語の形式的意味を表す LOTOS 仕様に対して直接形式的検証が行なえるという利点がある．

代数的仕様記述法を用いた UI 設計とプロトタイプ自動生成の先行研究に [16] がある．しかし，仕様記述言語の意味定義外にある副作用を利用して仕様記述を行なっており，代数的仕様記述法の利点である意味定義の簡明さが失われている．一方本研究では，代数的仕様本来の意味定義から逸脱する解釈は一切行なっていない．[19] では，ICO(Interactive Cooperation Objects) と呼ばれる，拡張されたペトリネットに基づく仕様記述法を提案している．ICO 仕様はペトリネットインタプリタにより直接実行されるのに対し，本研究で扱う ASM 仕様は，コンパイラにより比較的効率の良いプロトタイプに変換される．また，ICO 仕様では設計対象 UI と widget(UI 部品) の固定された 2 層構造しか許されない．一方，ASM 仕様では UI 間の親子関係は任意の深さで定義でき，また，オブジェクトの生成消滅のタイミングも仕様で自由に指定できる．

第2章　タスクモデルを用いたインタラクティブシステム設計法

2.1　はじめに

本章では，設計の上流工程から適用できる，体系化されたインタラクティブシステムの設計法を提案する．本設計法では，本研究で提案するタスク図を用いてタスクモデルを作成し，それを初期仕様，すなわち一連の設計プロセスへの入力とする．タスク図はユーザタスクの構造を表す図的言語である．設計プロセスにおいては，前段階の仕様を満たすような変換に基づいて仕様を詳細化する．タスク図の意味は仕様記述言語LOTOSによって形式的に定義される．具体的には，タスク図からその意味を定義するLOTOS仕様への変換規則を与える．次に，この変換規則にしたがってタスクモデルをLOTOS仕様に変換し，モデル検査法を用いてLOTOS仕様を形式的に検証する手順を示す．また，タスクモデルにUI設計情報を追加することにより，プロトタイプ自動生成を実現する方法についても示す．

本研究では,「通信販売による本の購入」を例題としてタスクモデルの記述，形式的検証やプロトタイプ自動生成を行なった結果について詳細に説明し，本設計法の有効性を確認する．

2.2　設計法の概略

本設計法の概略は図2.1のようになる．ここで，矩形は作成されるドキュメント，楕円は新しいドキュメントを作るための処理を表す．矢印は，各ドキュメントがどの処理の入出力となるかを示す．

以下，書籍の通信販売システムの設計を例題として説明を行う．

本設計法は大きく3つのフェーズに分けられる.「タスクモデルの作成」では，問題定義を記述し，それに基づいて本設計法において主たるシステム仕様となるタスクモデル

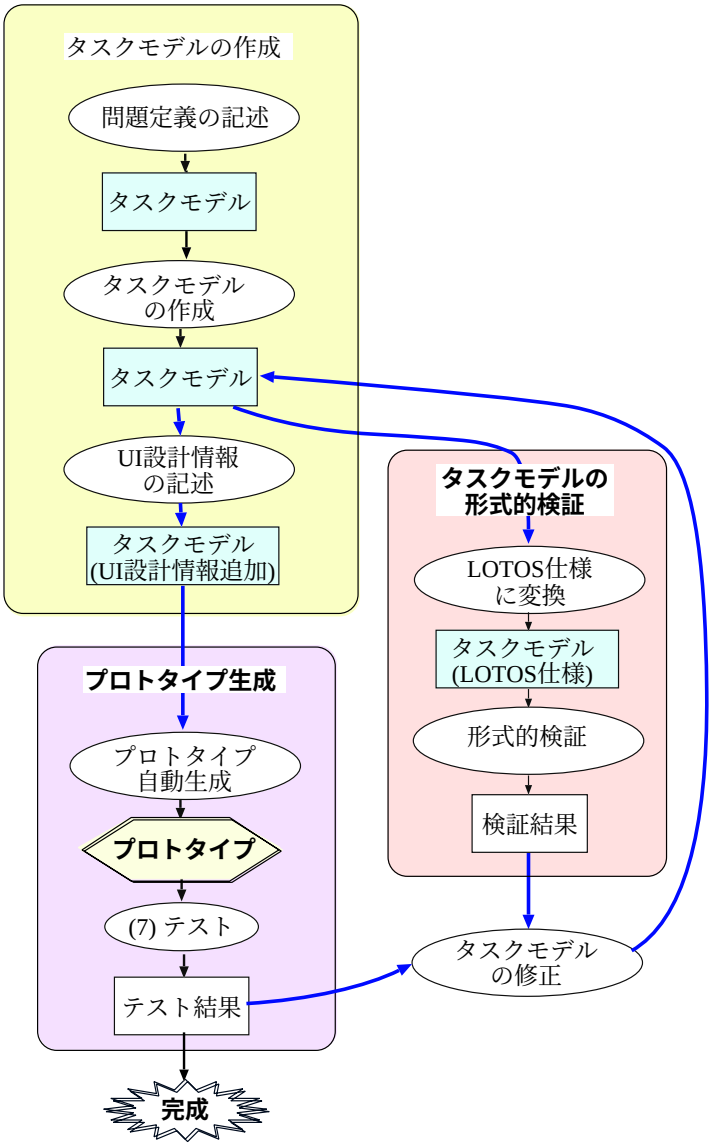

タスクモデルの作成
問題定義の記述
タスクモデル
タスクモデル
の作成
タスクモデル
UI設計情報
の記述
タスクモデル
(UI設計情報追加)
タスクモデルの
形式的検証
LOTOS仕様
に変換
タスクモデル
(LOTOS仕様)
形式的検証
検証結果
プロトタイプ生成
プロトタイプ
自動生成
プロトタイプ
(7) テスト
テスト結果
タスクモデル
の修正
完成

図 2.1 提案する設計法

を記述する．そして，タスクモデルに現実の入出力操作や操作画面との対応に関する情報（UI 設計情報）を追加する.「形式的検証」では，タスクモデル中の不具合を見つけるために形式的検証を行い，問題があればタスクモデルを修正し，再び形式的検証を行なう．検証すべき性質としてここでの例では，本を購入できるかどうか，本をキーワードで探した後分野でも探すことができるかなどが考えられる．形式的検証を行なって不具合を修正し十分に洗練した後,「プロトタイプ生成」において，UI 設計情報を追加したタスクモデルからプロトタイプの自動生成を行う．プロトタイプを使ったテストにより問題が見つかればタスクモデルを修正する．以降，タスクモデルの修正と形式的検証，および，プロトタイプの生成とそれを使ったテストを交互に繰り返す．最終的に，テストに合格したプロトタイプをアプリケーションのインタフェース部分とすることで一回の設計手順が完了する．

各処理で行う作業は以下のようになる．

タスクモデルの作成

(1) 問題定義を記述する．
問題定義（problem statement）[1] とは，設計するシステムの目的や必要とされる機能，対象とするユーザなどを自然言語で簡潔に記述したものである．

(2) タスクモデルを作成する．
タスクモデルとは，タスクとサブタスクの関係や実行手順などのタスクの構造を記述したものである．本設計法ではタスクモデルをシステムの主たる仕様として扱い，形式的検証などを通じてタスクモデルを洗練させること，および，タスクモデルに基づいてプロトタイプを生成することを設計の主な内容とする．タスクモデルの記述には，設計を通じて一貫して，タスク図という形式的記述法を用いる．タスク図については 2.5 節で詳しく述べる．

このステップでは，各種のタスク分析手法 [1, 20, 21] を使って，計算機システム化の対象であるタスクを分析しその構造を記述する．タスク構造の概要を記述した後，各タスクをサブタスクへ分解していくことで，タスクモデルを段階的に詳細化していく．本研究では，タスクモデルにおける最小単位は**イベント**であると考える．イベントとはユーザとシステムの間で行われる 1 回の通信（e.g., キーワードを指定する，一覧から本を選ぶ）である．上記の詳細化は，タスクの階層構造の葉が全てイベントになるまで行われる．詳しくは 2.4 節で述べる．

(3) UI 設計情報の記述を行う．
タスクモデル中の各イベントについて，現実の入出力操作や操作画面との対応に関する情報（UI 設計情報）を記述する．

タスクモデルの形式的検証

(4) タスクモデルの形式的検証を行う．
タスクモデルが満たすべき性質を時相論理式で記述し，作成されたタスクモデルがその性質を満たすかどうかを形式的に検証する．検証結果に基づいて，必要ならばタスクモデルを更新する．このステップについては 2.6 節で述べる．

プロトタイプ生成

(5) プロトタイプを自動生成する．
UI 設計情報を追加したタスクモデルに基づきプロトタイプの自動生成を行う．そして，プロトタイプを用いてテストを行う．操作手順とともに，UI 部品の配置，色，大きさや，表示される文言なども評価する．テスト結果に基づいて，必要ならばタスクモデルを更新する．ステップ (3) と (5) については 2.7 節で述べる．

2.3 問題定義の作成

問題の定義を自然言語で記述する．すなわち，設計するシステムの目的や必要とされる機能，対象とするユーザ，利用する場面等の制約などを記述する．この例では以下のようになる．

- ユーザが目的の本を見つけ，注文し，入手できるような計算機システムを構築する．
- 対象とするユーザは，WWW を利用でき，通信販売を利用したことがあるとする．

2.4 タスクモデルの作成

2.4.1 タスクモデル

本設計法では，問題定義をもとにタスク分析を行ない，抽出されたタスク構造（タスクモデル）をシステムの主な仕様として取り扱う．すなわち，システム化の対象であるユー

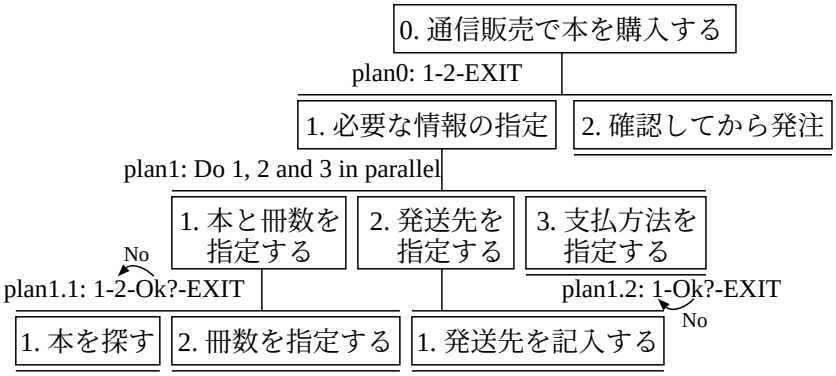


図 2.2 階層的タスク記述

ザタスクの流れを，タスク図という記述法を用いてタスクモデルとして記述する．タスク図については 2.5 節で述べる．タスク分析手法として階層的タスク分析（HTA）[22, 20] がよく知られている．HTA では，図 2.2 のような図的な記法で，タスクの構造，すなわちどのようなサブタスクをどのような順序で行うことで各タスクが実現されるかを記述していく．本研究でも HTA の考え方に基づき，タスク構造を（直観的にわかりやすく）記述することを，タスクモデル作成の主目的とする．

ここでサブタスクの実行順序をより明確に書き表すために,「階層的なフローチャート」の導入を考える.「階層的なフローチャート」とは，矢印によって実行順序を表し，かつ，各頂点（サブタスク）の定義も同じく「階層的なフローチャート」によって与えることができるものである．このような記法として，UML (Unified Modeling Language)[4] において定義されているアクティビティ図がある．本研究で用いるタスク図はアクティビティ図に基づいた記述法である．

タスク図の形式的意味は，プロセス代数に基づく記述言語である LOTOS[5, 6] を使って定義する．すなわち，任意のタスク図に対して，その意味を表す LOTOS 仕様を定義する (2.5.2 節)．LOTOS 仕様の意味はラベル付き遷移システム（LTS）で表され，それに対して形式的検証が可能である．タスクモデルの形式的検証については 2.6 節で詳しく述べる．

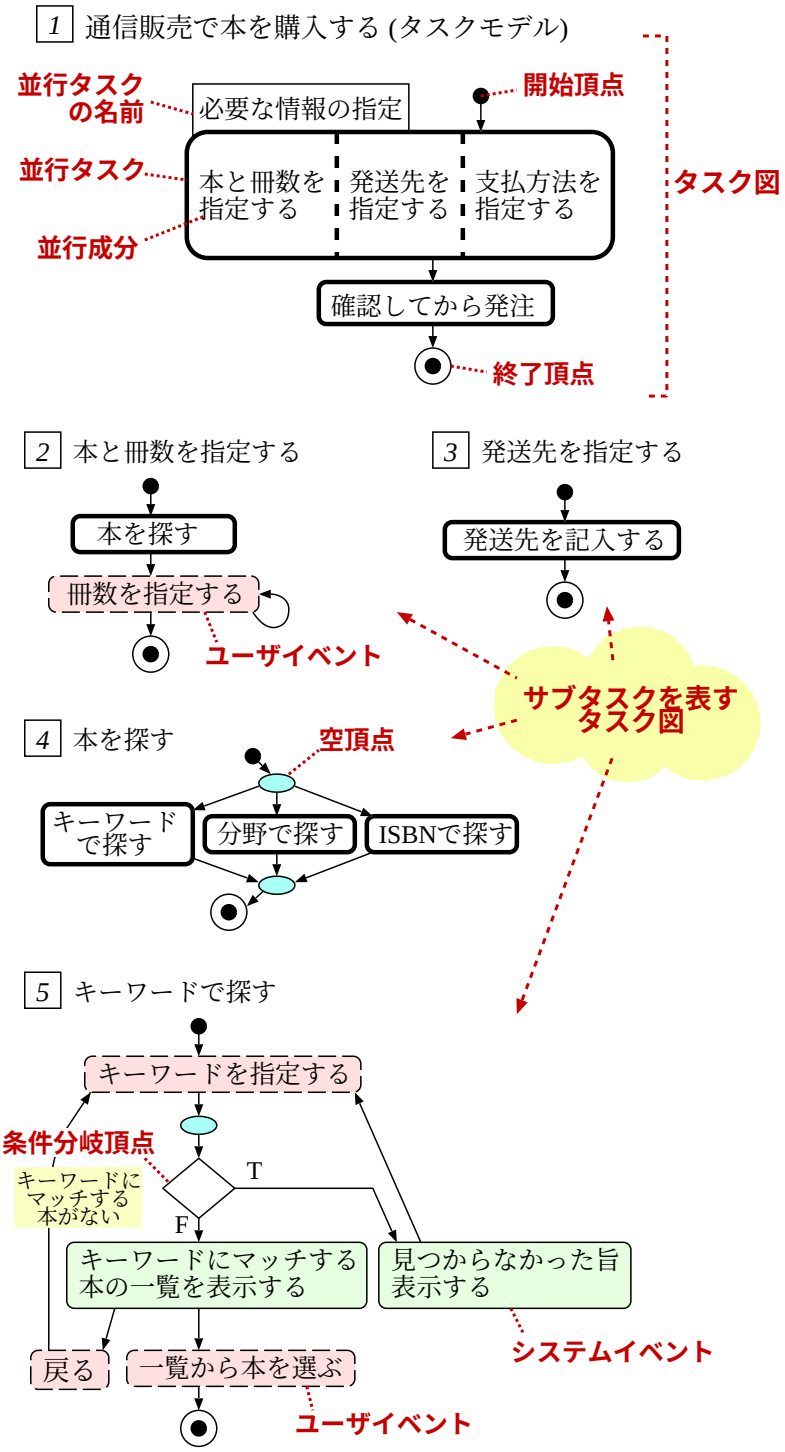

1 通信販売で本を購入する (タスクモデル)
並行タスクの名前
必要な情報の指定
開始頂点
並行タスク
本と冊数を指定する
発送先を指定する
支払方法を指定する
並行成分
タスク図
確認してから発注
終了頂点
2 本と冊数を指定する
本を探す
冊数を指定する
ユーザイベント
3 発送先を指定する
発送先を記入する
サブタスクを表すタスク図
4 本を探す
空頂点
キーワードで探す
分野で探す
ISBNで探す
5 キーワードで探す
キーワードを指定する
条件分岐頂点
キーワードにマッチする本がない
T
F
キーワードにマッチする本の一覧を表示する
見つからなかった旨表示する
システムイベント
戻る
一覧から本を選ぶ
ユーザイベント

図 2.3 タスクモデル

2.4.2 タスクモデルの詳細化

タスク図によるタスクモデルの記述例を図 2.3 に示す．この図では，タスク「通信販売で本を購入する」は，サブタスク「必要な情報の指定」,「確認してから発注」を順に実行することで，また「必要な情報の指定」はさらに 3 つのサブタスクの並行実行で実現できることが表現されている．

設計手順としては，まずタスクの概略を記述した後，原子タスク（それ以上分解されていないタスク）を複数のサブタスクに分解したり，新しいタスクや計算結果に基づく条件分岐を追加したりすることで，より詳細なタスクモデルを作成する．すなわち，タスク図はタスクを階層的に表現できるようになっている

詳細化と並行して，タスクモデルには，タスク構造に加え，タスク内で扱われるデータに関する記述，すなわち，データ属性の記述を行う．各イベントにおいては，ユーザとシステムの間でやり取りされるデータをデータ属性として記述する．図 2.3 ではタスクのデータ属性が記述されていないが，例えば，イベント「キーワードを指定する」においては，キーワードを表す任意の文字列がユーザからシステムに送信される．イベント以外のタスクにおいては，それ以前に実行されたタスクから受け取るデータ（入力データ），およびそれ以降に実行されるタスクへ渡されるデータ（出力データ）について記述する．これらは各イベントで行われる内容を明確にする働きがあり，特にプロトタイプを生成するためには必須の情報となる．データ属性の記述については 2.4.3 節で詳しく述べる．

また，タスクモデル作成の初期段階においてはタスク間のデータ依存関係を求めることでタスク構造の概要を獲得することができる．

ユーザとシステム間でのデータのやりとりは UI 部品を介して行われる．2.7.1 節では，各タスクで使用する UI 部品などの UI 設計情報の記述について述べる．

データの依存関係に基づくタスクモデルの獲得

各タスクの入出力データ間の依存関係に基づき，タスクの実行順序を決める．これにより，データに依存関係のないタスク間に対して不必要に実行順を指定することを防ぐことができる．

(1) タスクを列挙する．

(2) 各タスクが参照したり更新したりするデータを列挙する．タスク t が参照／更新するデータを $t.d$ という形式で書く．これを t に属するデータと呼ぶ.

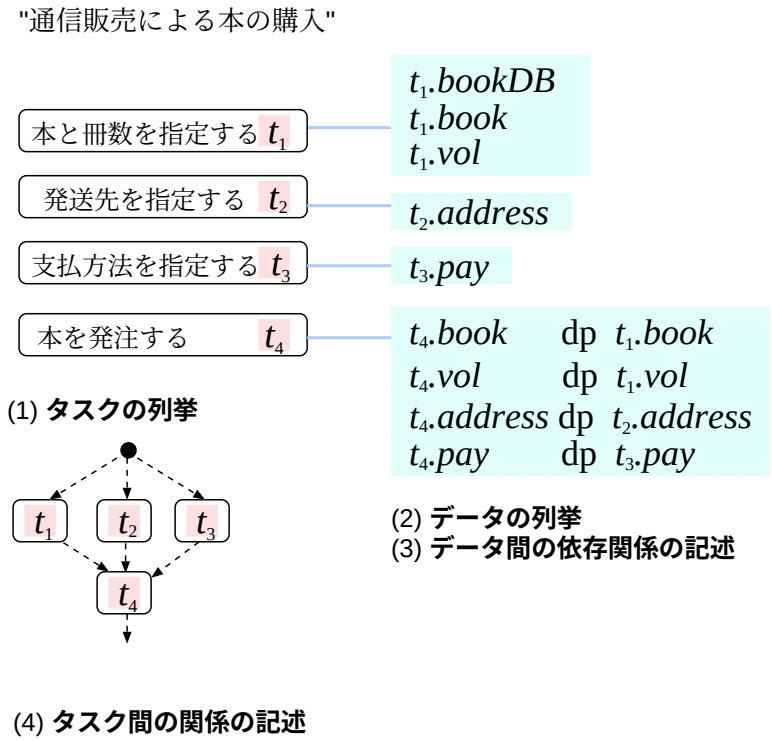


図 2.4　タスクモデルの基本設計

(3) 異なるタスクに属するデータ間の依存関係を記述する．データ $t_1.d_1$ がデータ $t_2.d_2$ に依存しているとき，

$$t_1.d_1 \ \mathrm{dp}\ t_2.d_2$$

と記述する．図 2.4(3) において，タスク「本を発注する」に属するデータ $t_4.book$ は，タスク「本と冊数を指定する」に属するデータ $t_1.book$ に依存して決まる（$t_1.book$ そのものである）．したがって，$t_4.book \ \mathrm{dp}\ t_1.book$ と記述する．

(4) タスク間の関係 $\prec$ を，次の条件を満たす最小の関係と定義する．

- あるデータ $t_1.d_1, t_2.d_2$ に対して，$t_1.d_1 \ \mathrm{dp}\ t_2.d_2$ ならば，$t_2 \prec t_1$.
- $t_1 \prec t_2$ かつ $t_2 \prec t_3$ ならば $t_1 \prec t_3$.

上で定義された $\prec$ は半順序であると仮定する（すなわち，どの t についても $t \prec t$ とはならない）．

(5) (4) で定義した関係 $\prec$ に基づき，次の基本操作を繰り返してタスク図を作成する．

- あるタスク t_1 から別のタスク t_2 へ有向辺を加える（t_1 が start，または，t_2 が done の場合を含む）.
- 複数のタスク $t_1, t_2, \cdots, t_n$ をまとめて，それらを並行成分とする並行タスク t を作る．ただし，どの $i, j (1 \leq i, j \leq n)$ についても $t_i \not\prec t_j$ のときに限る．
  任意の t' について，$t_i \prec t'$ となる i（$1 \leq i \leq n$）が存在するとき，$t \prec t'$ と定義する．また，$t' \prec t_i$ となる i（$1 \leq i \leq n$）が存在するとき $t' \prec t$ と定義する．

ただし，上の操作の結果得られた有向グラフ G はタスク図であるための条件（付録の定義 1）と次の条件を満たさなければならない.

- $t \prec t'$ ならば，t から t' へのパスが G に存在すること.

図 2.4(4) のタスク間の関係の記述を参照することにより，図 2.3 でのタスク「通信販売による本の購入」は,「本と冊数を指定する」「発送先を指定する」「支払方法を指定する」の 3 タスクの並行実行後, タスク「本を発注する」を実行するように設計されている.

イベントの指定　タスクモデルは，ユーザとシステムの両者を成分に持つ，協調システムの仕様を与えていると考えることができる．つまり，両者が通信し合って作業を行うことで，記述されたタスクが実現される．本設計法では，タスクモデルにおける最小単位はイベントであると考える．イベントとはユーザとシステムの間で行なわれる 1 回の通信である．そこで，タスクの階層構造の葉が，ユーザからシステムへの 1 回の通信（ユーザイベント）またはシステムからユーザへの 1 回の通信（システムイベント）になるまでタスクモデルの詳細化を行なう.

- ユーザイベントの例 : 「キーワードを指定する」
- システムイベントの例 : 「キーワードにマッチする本の一覧を表示する」

2.4.3　データ属性の記述

2.4.2 節において，データの依存関係を利用してタスクの実行順を決める方法を述べた．この節では各データの入出力モードやソートなどの情報を含むデータ属性の記述について述べる.

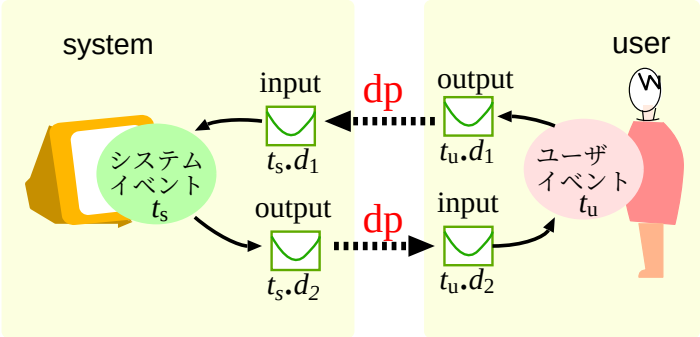


図 2.5 データの種類

表 2.1 データの入出力モード

データの種類	入出力モード
合成タスクが出力するデータ	TaskOutput
合成タスクの入力となるデータ	TaskInput
＜ユーザイベント＞ ユーザからシステムへ送信されるデータ ＜システムイベント＞ システムからユーザへ送信されるデータ	output
送信される値を得るために必要なデータ	input

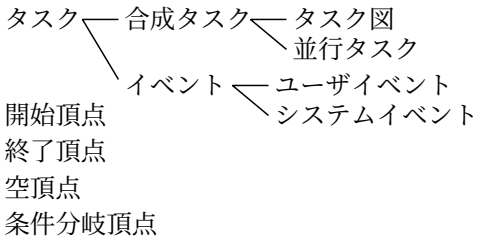
タスクで扱うデータの分類

タスクやイベントで参照したり変更したりする各データを表 2.1 に示す 4 種類に分類し，各データに対して入出力モードを指定する（図 2.5）．

2.4.2 節で述べたとおり，タスク t に属するデータは $t.d$ という名前で表され，異なるタスク間では共有されない．タスク間でのデータの授受は依存関係 dp で表現される．

ソート

タスクモデル中で使われている各ソートは，抽象データ型 [6] の形で別途定義されているとする．ここでは，Integer，String などの単純組み込み型，設計者が定義する型（図 2.7 での Book），および，ソート T の要素の集合を表す SetOf$\langle T \rangle$ 型（図 2.7 での SetOfBook）などが定義されているものとする．



実線は，左にある語が右にある語の上位概念であることを示す．

図 2.6　タスク図の頂点

データ属性の構文

タスク図では，図 2.6 に示す頂点を用いてタスク構造を表現する．すべてまたは一部の頂点に，その頂点の種類に応じて，以下のような構文でデータ属性を指定できる．

データ属性は，イベントにおいてはそのイベントで送受信される値，タスクにおいてはそれ以前のタスクから受け取ったり以降のタスクに渡したりする値を記述したものである．また，空頂点においては特定の値をデータに束縛することができ，その指定もデータ属性と呼ぶ．条件分岐頂点における条件を表す式もデータ属性と呼ぶ．

以下では backus naur 記法 [23] に基づく記法で構文を示す．ただし，“” で囲まれた文字列は終端記号を表し，それ以外の単語は非終端記号を表す．また，[ ] で囲まれた部分は 0 回または 1 回の出現を表し，{ }∗ は 0 回以上の出現を表す．本文中では非終端記号を ⟨⟩ で囲んで示す．

- データ

  データ ::= 変数 | 頂点名 “.” 変数

- 合成タスク

  データ属性 ::= { “TaskInput” 仮引数 “=” 入力項 “:” 入力ソート }∗
  { “TaskOutput” 出力変数 “=” 出力項 “:” ⟨出力ソート⟩ }∗
  仮引数 ::= データ
  出力変数 ::= データ
  入力項 ::= 項
  出力項 ::= 項
  入力ソート ::= ソート
  出力ソート ::= ソート

  以前のタスクからこのタスクに入力される値を ⟨入力項⟩ で，このタスクから以降のタスクに出力される値を ⟨出力項⟩ で指定する．入力された値は，このタスク内では ⟨仮引数⟩ で参照される．以降のタスクでは，出力された値を ⟨出力変数⟩ で参照できる．

- システムイベント

  データ属性 ::= { “input” データ “dp” 入力データ }∗
  { “output” データ [“=” ⟨項⟩] “:” ソート }∗
  入力データ ::= データ

  ⟨入力データ⟩ には他のタスクに属するデータを，それ以外にはこのイベントに属するデータを記述する．input では，このシステムイベントで参照するデータを ⟨入力データ⟩ で指定する．これによりどのタスクの データ を参照するかを明示できる．output では，このシステムイベントにおいてシステムが送信するデータを指定する．⟨項⟩ はこのイベントにおいてシステムから送信される値を表す．⟨項⟩ が省略された場合は ⟨データ⟩ の値が送信される．⟨ソート⟩ は送信される値のソートを表す．

- ユーザイベント

  データ属性 ::= { “input” データ “dp” 入力データ }∗
  { “output” データ [関係 項] “:” ソート }∗
  関係 ::= “∈” | “⊆”
  入力データ ::= データ

⟨入力データ⟩には他のタスクに属するデータを，それ以外にはこのイベントに属するデータを記述する．input では，このユーザイベントで参照するデータを⟨入力データ⟩で指定する．これによりどの データ を参照するかを明示できる．output では，このユーザイベントにおいてユーザが送信するデータを指定する．“∈” ⟨項⟩ や “⊆” ⟨項⟩ は送信される値に対する制約を表す．これらはそれぞれ，複数の選択肢からひとつが選択されるイベント，複数の選択肢から 0 個以上が選択されるイベントにおいて用いられる（e.g., 本の集合から本を選ぶ）．実際に送受信される値はこの制約を満たす範囲で任意に選択される．任意の値が送信されるイベント（e.g., キーワードを指定する）では，⟨関係⟩⟨項⟩ が省略される．⟨ソート⟩ はデータのソートを表す．システムイベントの場合と同様に，送信された値は，以降のタスクでは ⟨データ⟩ で参照できる．

- 空頂点

  データ属性 ::= “let” データ “=” 項 “:” ソート

  項 の値を データ に束縛する．

- 条件分岐頂点

  データ属性 ::= 条件

  条件 ::= 等式 | ブール式

  ⟨条件⟩ が真のときはラベル T の付いた辺，偽のときはラベル F の付いた辺の先に遷移する．

- 開始頂点および終了頂点はデータ属性を持たない．

データ属性の記述例

図 2.7 は，タスク「本と冊数を指定する」とタスク「キーワードで探す」の各タスクにデータ属性を記述した例である．ここでは簡単のため，タスク名を t_x（x は整数）と略記し説明する．依存関係 dp にある 2 つのデータは同じソートであるとする．ユーザイベント「一覧から本を選ぶ t_{54}」では，システムイベントにおいて送信されたデータ $t_{52}.blist$ をこのユーザイベントが参照しデータ $t_{54}.blist$ に格納すること，データ $t_{54}.blist$ から 1 つ要素を選択しデータ $t_{54}.b$ に格納し，それをシステムに送信することが記述されている．0 個以上データを選択する場合は，$\in$ の代わりに $\subseteq$ を用いてデータ間の関係を記

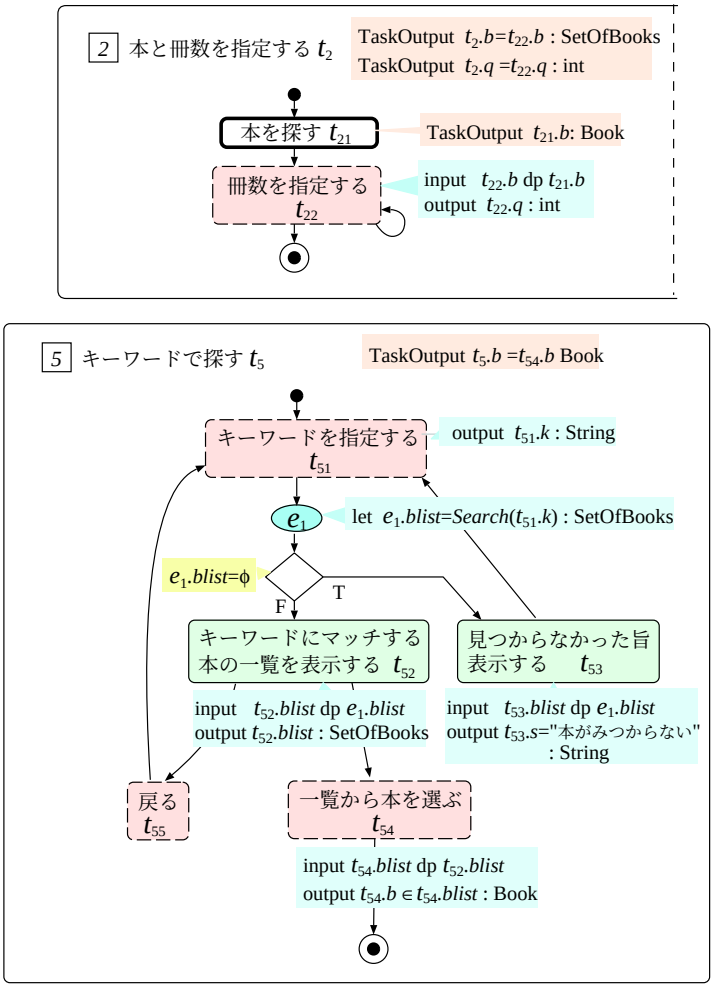


図 2.7 データ属性の記述

述する．また，システムイベント「見つからなかった旨表示する」のデータ属性 output: $t_{53}.blist$ ＝“本が見つからない” : String のように，各システムイベントにおいて送信される値を記述することができる．

また，空頂点 e_1 のデータ属性の項は，この空頂点より以前のタスクのデータ属性で記述されているデータ（ここではタスク t_{51} のデータ $t_{51}.k$）を使って指定する．

2.5 タスク図の定義

タスクモデルの記述およびその後の詳細化のために，タスク図という図的言語を導入する．ここでは，タスク図の構文および意味の定義を行なう．タスク図の意味は，タスク図から LOTOS 仕様に変換するための規則によって定義される．

2.5.1 タスク図の構文

定義 1 (タスク図) タスク図は，有向グラフ (V, A) である．頂点 $v \in V$ は図 2.6 に挙げたもののうちのいずれかである．ただし，開始頂点，終了頂点はそれぞれちょうど 1 個 V に含まれる．開始頂点の入次数および終了頂点の出次数は 0 である．各頂点 $v \in V$ について，開始頂点から v へのパスおよび v から終了頂点へのパスが存在する．

条件分岐頂点から出る辺はラベル T または F を持つ．それ以外の辺はラベルを持たない．

全てまたは一部の頂点はそれぞれデータ属性を持つ． □

定義 2 (タスク) ユーザイベントおよびシステムイベントをイベントと呼ぶ．タスク図および並行タスクを合成タスクと呼ぶ．イベントおよび合成タスクをタスクと呼ぶ（図 2.6）． □

定義 3 (タスクモデル) タスクモデルはタスク図で与えられる． □

タスク図において，各頂点は全体のタスクを構成するサブタスクを表し，辺はその実行順序を表している．すなわちタスク図は，開始頂点から終了頂点までの任意のパス（同じ頂点を複数回含んでもよい）について，そのパスに沿ってサブタスクを行うことで全体のタスクが実現できることを表している．

タスクの最小単位はイベントである．イベントとは，ユーザとシステムとの間で行われる 1 回の通信のことである．送信者がユーザであるかシステムであるかに応じて，それぞれユーザイベント，システムイベントと呼ぶ．例えば「キーワードを指定する」はユーザイベントの例,「キーワードにマッチする本の一覧を表示する」はシステムイベントの例である．一方，例えば「本を検索する」という処理を考えると，これは 1 回の通信を表すものではないのでイベントではない．タスクモデルを記述する際は，検索手続きなど，通信ではない動作を無視して書くこととする（cf. 図 2.3「キーワードで探す」）．

定義 4 (並行タスク) 並行タスクは，タスクを要素とする空でない集合である． □

並行タスクは，複数の並行成分からなるタスクを表す．図中では並行成分を破線で区切って表す．

空頂点は，辺の数を減らして図を見やすくするために用いられる．また，後で述べるようにデータ属性を指定することで，イベント以外の箇所で特定の値を変数に束縛するためにも用いることができる．図中では，中が空白の楕円で表される．図 2.3 において，タスク「本を探す」は，空頂点を 2 個含んでいる．

条件分岐頂点は，特定の式が真か偽かに応じて遷移先を変えるために用いられる．

階層構造

タスク図や並行タスクを頂点とすることでタスクの階層構造を表現できる．図 2.3 のタスク「本と冊数を指定する」は,「本を探す」「冊数を指定する」を頂点とするタスク図である．

定義 5 (サブタスク) タスク図 t に含まれる頂点であるタスクを t のサブタスクと呼ぶ．並行タスク t' の成分を t' のサブタスクと呼ぶ． □

定義 6 タスク t に対し，t の頂点または成分の集合 $V[t]$ を以下のように定義する．

$$V[t] = \begin{cases} V_t & if\, t = (V_t, A_t) \text{がタスク図} \\ t & if\, t \text{が並行タスク} \\ \emptyset & otherwise. \end{cases}$$

t が並行タスクのとき，定義 4 より t 自身がタスクの集合であることに注意．タスク図 t の辺集合を $A[t]$ と書く． □

定義 7 (子孫タスク) タスク t' が t のサブタスクであるとき，すなわち $t' \in V[t]$ のとき，$t' \sqsubset t$ と書く．ある整数 $k \geq 1$ についてタスク $t'_0, \ldots, t'_k$ が存在し，

$$t' = t'_0 \sqsubset t'_1 \sqsubset \cdots \sqsubset t'_{k-1} \sqsubset t'_k = t$$

であるとき，$t' \overset{+}{\sqsubset} t$ と書く．

タスク t に対し，子孫タスクの集合 $V^*[t]$ を以下のように定義する．

$$V^*[t] = \{t' \mid t' \overset{+}{\sqsubset} t \text{ または } t' = t\}.$$

□

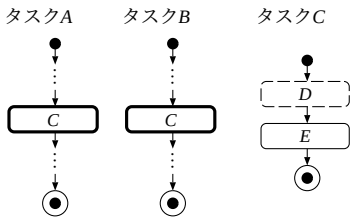


図 2.8 サブタスクの共有

ここで，タスクとサブタスクの関係は木構造を構成するものとする．つまり，自分自身を内部に含むタスクや，互いに要素を共有するようなタスクは存在しないとする．形式的に書くと以下のようになる．

仮定 8 どのタスク t についても，$t \overset{+}{\sqsubset} t$ ではない．任意の異なるタスク t，t' について，$V[t] \cap V[t'] = \emptyset$ である． □

図 2.2 のようなタスクの階層構造を分析・記述する際に,「タスク t を実現する過程の一部に t 自身の実行が含まれる」といった再帰的な構造が現れることは稀であると考えられる．そこで，定義を繁雑にしないために，自分自身をサブタスクまたはサブタスクの子孫タスクとすることは許さないものとした．一方，図 2.8 のように，複数のタスクが同じサブタスクを共有することは，記述を簡潔にするのに役立ち，実際このように記述しても構わない．ただし形式上は，タスク A における頂点 C とタスク B における頂点 C は異なる有向辺に接続する異なる頂点であるので，この 2 頂点は内部が同じである異なる頂点として扱う．

2.5.2 タスク図の LOTOS への変換

タスク図の意味は，プロセス代数に基づく記述言語である LOTOS[6] を使って定義する．

以降，タスクモデル tm を固定し，tm の意味を表す LOTOS 仕様 $Spec$ を定義する．LOTOS 仕様の意味は 1 個のラベル付き遷移システム（LTS）で表される [6]．すなわち，tm の意味は $Spec$ の意味である LTS と等しい．データ属性に関して次のように仮定する．

- tm 自身はデータ属性を持たない．
- tm 以外の合成タスクは，データ属性に “TaskInput” 部分と “TaskOutput” 部分がそれぞれちょうど 1 個ずつ指定されているとする．

変数の有効範囲について述べる．合成タスクの中で使われる変数はすべてそのタスクの局所変数とする．タスク図 t の局所変数は，t の仮引数，t の頂点であるイベントや空頂点（ただし，変数のみ指定されているシステムイベントを除く）のデータ属性に指定されている変数，および t の頂点である合成タスクの出力変数からなる．t 中で使われる変数，すなわち，t の頂点のデータ属性中で使われている変数（合成タスクである頂点の仮引数および出力項を除く）および t の出力項で使われている変数は，t の局所変数でなければならない．これにより，合成タスクと他のタスクの間での値の受渡しは，必ず仮引数・出力変数を通じて行われることとなる．

並行タスク t' の局所変数は，t' の仮引数，t' の成分であるイベント（ただし，変数のみ指定されているシステムイベントを除く）に指定されている変数，および t' の成分である合成タスクの出力変数からなる．タスク図の場合と同様に，t' 中で使われる変数は t' の局所変数でなければならない．また，t' の複数の成分が同じ出力変数を持ってはいけない．

準備として，次のように各記号を定義する．タスク図 t の開始頂点を start_t，終了頂点を done_t と書く．タスク図 t と頂点 $v \in V[t]$ に対して，$V[t]$ の部分集合 $N(t,v)$，および，v が条件分岐頂点のときはさらに $NT(t,v)$, $NF(t,v)$ を，以下のように定義する．これらは頂点 v からの遷移先の集合を表す．

$$
\begin{aligned}
N(t,v) &= \{v' \in V[t] \mid (v,v') \in A[t]\},\\
NT(t,v) &= \{v' \in V[t] \mid (v,v') \in A[t] \text{ かつ } (v,v') \text{ のラベルが T}\},\\
NF(t,v) &= \{v' \in V[t] \mid (v,v') \in A[t] \text{ かつ } (v,v') \text{ のラベルが F}\}.
\end{aligned}
$$

合成タスク t に対して，t の子孫タスクであるイベントの集合 $events(t)$ を以下のように定義する．

$$events(t) = \{t' \in V^*[t] \mid t' \text{ がイベント }\}.$$

合成タスク t に対し，名前 $\mathrm{Sname}(t)$, $\mathrm{Uname}(t)$ が与えられるとする．これらはそれぞれプロセス名として用いられる．プロセス $\mathrm{Sname}(t)$ は t におけるシステムの動作，プロセス $\mathrm{Uname}(t)$ はユーザの動作を表すものとなる．イベント t' に対しては名前 $\mathrm{Name}(t')$ が与えられるとする．これはゲート名として用いられる．タスク図 t の頂点 $v \in V[t]$ に対し，上記とは別に，名前 $\mathrm{Vsname}(v)$,$\mathrm{Vuname}(v)$ が与えられるとする．これらもプロセス名として用いられる．プロセス $\mathrm{Vsname}(v)$ はプロセス $\mathrm{Sname}(t)$ の中で，プロセス $\mathrm{Vuname}(v)$ はプロセス $\mathrm{Uname}(t)$ の中で補助的に用いられ，v の内容を実行した後，次

の頂点に遷移するようなプロセスとなる．異なるタスク・頂点同士はこれらの名前を共有しないものとする．

以下では 2.4.3 節と同様の記法を使って，tm の意味を表す LOTOS 仕様を定義する．すなわち，$\langle \mathrm{Spec} \rangle$ を開始記号として導出される記号列（tm に対して一意に決まる）が目的の LOTOS 仕様である．ただし，ここでは $\langle \mathrm{Sproc}(t) \rangle$ のように非終端記号が引数を取れるものとする．各規則中の変数が取り得る値は，規則の末尾の *if* 節や *where* 節で制限される．“␣” は空文字列を表す．規則 (2.1) からわかるように，導出される LOTOS 仕様は，プロセス $\mathrm{Sname}(tm)$ と $\mathrm{Uname}(tm)$ の完全同期並列合成として表される．

$$
\begin{aligned}
\mathrm{Spec} \quad ::= \quad & \text{“specification” “}\mathit{Spec}\text{” } \mathrm{Gates}(tm) \text{ “:” “exit”} \\
& \text{“behaviour” } \mathrm{Sname}(tm)\ \mathrm{Gates}(tm) \text{ “}\|\text{” } \mathrm{Uname}(tm)\ \mathrm{Gates}(tm) \\
& \text{“where” } \mathrm{Sproc}(tm)\ \mathrm{Uproc}(tm) \text{ “endspec”} && (2.1) \\
\mathrm{Gates}(t) \quad ::= \quad & \text{“[” } \mathrm{Name}(v_1) \text{ “,” } \cdots \text{ “,” } \mathrm{Name}(v_n) \text{ “]”} \\
& \quad \textit{if } t \text{ が合成タスク．} \mathit{events}(t) = \{v_1, \ldots, v_n\} && (2.2) \\
\mathrm{Sproc}(t) \quad ::= \quad & \text{“process” } \mathrm{Sname}(t)\ \mathrm{Gates}(t) \text{ “(” 仮引数 } (t) \text{ “:” 入力ソート } (t) \text{ “)”} \\
& \text{“:” “exit” “(” 出力ソート } (t) \text{ “)” “:=” } \mathrm{Pbody}(t) \\
& \text{“endproc” } \mathrm{Paux}(t) \qquad \textit{if } t \text{ が合成タスク} && (2.3) \\
\mathrm{Sproc}(t) \quad ::= \quad & \text{“␣”} \qquad \textit{otherwise} && (2.4) \\
\mathrm{Pbody}(t) \quad ::= \quad & \mathrm{Vpcall}(t, \mathrm{start}_t) \qquad \textit{if } t \text{ がタスク図} && (2.5) \\
\mathrm{Pbody}(t) \quad ::= \quad & \mathrm{Parbody}(t, v_1) \text{ “}|||\text{” } \cdots \text{ “}|||\text{” } \mathrm{Parbody}(t, v_n) \\
& \text{“}\gg\text{accept” 出力変数 } (v_1) \text{ “:” 出力ソート } (v_1) \text{ “,” } \cdots \text{ “,”} \\
& \text{出力変数 } (v_n) \text{ “:” 出力ソート } (v_n) \text{ “in” “exit(” 出力項 } (t) \text{ “)”} \\
& \quad \textit{if } t = \{v_1, \ldots, v_n\} \text{ が並行タスク} && (2.6) \\
\mathrm{Paux}(t) \quad ::= \quad & \mathrm{Vsproc}(t, v_1) \cdots \mathrm{Vsproc}(t, v_n) \\
& \quad \textit{if } t = (V_t, A_t) \text{ がタスク図．} V_t - \{\mathrm{done}_t\} = \{v_1, \ldots, v_n\} && (2.7) \\
\mathrm{Paux}(t) \quad ::= \quad & \mathrm{Sproc}(v_1) \cdots \mathrm{Sproc}(v_n) \\
& \quad \textit{if } t = \{v_1, \ldots, v_n\} \text{ が並行タスク} && (2.8) \\
\mathrm{Vsproc}(t, v) \quad ::= \quad & \text{“process” } \mathrm{Vsname}(v)\ \mathrm{Gates}(t) \text{ “(” 局所変数宣言 } (t) \text{ “)”} \\
& \text{“:” “exit” “(” 出力ソート } (t) \text{ “)” “:=” } \mathrm{Vbody}(t, v) \\
& \text{“endproc” } \mathrm{Sproc}(v) && (2.9) \\
\mathrm{Vbody}(t, v) \quad ::= \quad & \text{“[” 条件 } (v) \text{ “]” “}\text{–}\rangle\text{” } \mathrm{Next}(t, v_1, \ldots, v_n) \\
& \text{“[]” “[not(” 条件 } (v) \text{ “)]” “}\text{–}\rangle\text{” } \mathrm{Next}(t, v'_1, \ldots, v'_m) \\
& \quad \textit{if } v \text{ が条件分岐頂点．} NT(t, v) = \{v_1, \ldots, v_n\}, \\
& \quad NF(t, v) = \{v'_1, \ldots, v'_m\} && (2.10) \\
\mathrm{Vbody}(t, v) \quad ::= \quad & \mathrm{Action}(v)\ \mathrm{Next}(t, v_1, \ldots, v_n) \\
& \quad \textit{otherwise.}\ N(t, v) = \{v_1, \ldots, v_n\} && (2.11) \\
\mathrm{Next}(t, v_1, \ldots, v_n) \quad ::= \quad & \text{“(” } \mathrm{Vpcall}(t, v_1) \text{ “[]” } \cdots \text{ “[]” } \mathrm{Vpcall}(t, v_n) \text{ “)”} && (2.12)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Vpcall}(t,v) \quad &::= \quad \text{“exit(” 出力項 } (t) \text{ “)”} \qquad \textit{if } v \text{ が終了頂点} && (2.13)\\
\mathrm{Vpcall}(t,v) \quad &::= \quad \text{“Vsname}(v)\text{” Gates}(t) \text{ “(” 局所変数初期化 } (t) \text{ “)”} \\
&\qquad \textit{if } v \text{ が開始頂点} && (2.14)\\
\mathrm{Vpcall}(t,v) \quad &::= \quad \text{“Vsname}(v)\text{” Gates}(t) \text{ “(” 局所変数 } (t) \text{ “)”} \qquad \textit{otherwise} && (2.15)\\
\\
\mathrm{Parbody}(t,v) \quad &::= \quad \text{“(” Action}(v) \text{ “exit(” Pardata}(v,v_1) \text{ “,” } \cdots \text{ “,” Pardata}(v,v_n) \text{ “)” “)”} \\
&\qquad \textit{if } t = \{v_1,\ldots,v_n\} \text{ が並行タスク} && (2.16)\\
\mathrm{Pardata}(v_1,v_2) \quad &::= \quad \text{出力変数 } (v_2) \qquad \textit{if } v_1 = v_2 && (2.17)\\
\mathrm{Pardata}(v_1,v_2) \quad &::= \quad \text{“any” 出力ソート } (v_2) \qquad \textit{if } v_1 \neq v_2 && (2.18)\\
\\
\mathrm{Action}(v) \quad &::= \quad \text{“let” 変数 } (v) \text{ “:” ソート } (v) \text{ “=” 項 } (v) \text{ “in” Name}(v) \\
&\qquad \text{“!” 変数 } (v) \text{ “;”} \qquad \textit{if } v \text{ がシステムイベント. データ属性に} \\
&\qquad\qquad\qquad\qquad \text{変数と項が指定されている} && (2.19)\\
\mathrm{Action}(v) \quad &::= \quad \text{Name}(v) \text{ “!” 変数 } (v) \text{ “;”} \\
&\qquad \textit{if } v \text{ がシステムイベント. データ属性に} \\
&\qquad \text{変数のみ指定されている} && (2.20)\\
\mathrm{Action}(v) \quad &::= \quad \text{Name}(v) \text{ “?” 変数 } (v) \text{ “:” ソート } (v) \text{ “;”} \\
&\qquad \textit{if } v \text{ がユーザイベント. データ属性を持つ} && (2.21)\\
\mathrm{Action}(v) \quad &::= \quad \text{Name}(v) \text{ “;”} \qquad \textit{if } v \text{ がイベント. データ属性を持たない} && (2.22)\\
\mathrm{Action}(v) \quad &::= \quad \text{Sname}(v) \text{ Gates}(v) \text{ “(” 入力項 } (v) \text{ “)” “⟫accept”} \\
&\qquad \text{出力変数 } (v) \text{ “:” 出力ソート } (v) \text{ “in”} \qquad \textit{if } v \text{ が合成タスク} && (2.23)\\
\mathrm{Action}(v) \quad &::= \quad \text{“let” 変数 } (v) \text{ “:” ソート } (v) \text{ “=” 項 } (v) \text{ “in”} \\
&\qquad \textit{if } v \text{ が空頂点. データ属性を持つ} && (2.24)\\
\mathrm{Action}(v) \quad &::= \quad \text{は以下の通りである. “␣”} \qquad \textit{otherwise} && (2.25)
\end{aligned}
$$

上記の中で，仮引数 (t)，出力変数 (t) などは，t のデータ属性に指定されている各要素を表す．局所変数宣言 (t) は，t の各局所変数名とそのソート名の組（ “:” で区切られる）の列（ “,” で区切られる）である．局所変数 (t) は t の局所変数名の列である．局所変数初期化 (t) は，局所変数 (t) と同様だが，t の仮引数以外の変数については，変数名の代わりにそのソートにおける任意の値（初期値）を置いた列である．

“let”，“?” または “⟫accept” を生成する規則（(2.6), (2.19), (2.21), (2.23), (2.24)）は，新しい変数を宣言するような動作式を生成する．宣言される変数が他の変数と干渉しないように，これらの規則による導出においては，タスク・頂点のデータ属性に指定された変数名そのものではなく新しい変数名（例えば末尾に “_1” を付けたものなど）を使う．同時に，同じ動作式（同じ $\mathrm{Vbody}(t,v)$ または $\mathrm{Parbody}(t,v)$ から導出された部分記号列）においてより右側にその変数が現れる場合，それも同じ名前に変える．すなわち，規則 (2.19), (2.21), (2.23), (2.24) の右辺中で変数名を変えた場合，同じ動作式に現れる

規則 (2.13) の 出力項 (t)，規則 (2.15) の 局所変数 (t)，規則 (2.17) の 出力変数 (v_2) においても変数名を変える．また，規則 (2.6) の右辺中で変数名を変えた場合，同じく規則 (2.6) の右辺中の 出力項 (t) においても変数名を変える（cf. 図 2.9–図 2.14）．

$\langle \mathrm{Uproc}(t) \rangle$ に対する規則およびそれに付随する規則は，規則 (2.3)–(2.25) と同様に定義する．ただし以下の点が異なる．

- $\mathrm{Sname}(t)\,, \mathrm{Vsname}(t)$ の代わりにそれぞれ $\mathrm{Uname}(t)\,, \mathrm{Vuname}(t)$ を使う．
- 条件分岐頂点 v に対して規則 (2.10) ではなく (2.11) を適用する．（ユーザは条件について関知しないことを表す．）
- データ属性を持つ空頂点 v に対して規則 (2.24) ではなく (2.25) を適用する．（変数への値の束縛はシステム内部で行われることを表す．）
- 規則 (2.19)–(2.21) を以下の規則 (2.26)–(2.27) に置き換える．（ユーザイベントにおいてはユーザとシステム双方が変数宣言 “?” を行うので，同期によって値の生成が行われる．）

$\mathrm{Action}(v)$ ::= $\mathrm{Name}(v)$ “?” 変数 (v) “:” ソート (v) “[” 変数 (v) 関係 (v) 項 (v) “]” “;”
if v がユーザイベント．データ属性に “depends_on” 以外の関係と項が指定されている (2.26)

$\mathrm{Action}(v)$ ::= $\mathrm{Name}(v)$ “?” 変数 (v) “:” ソート (v) “;”
if v がイベント．上記以外のデータ属性を持つ (2.27)

2.5.3 LOTOS 仕様の導出例

以上の規則によってどのような LOTOS 仕様が得られるか例を使って説明する．t をタスク図，$v, v_1, \ldots, v_n, v'_1, \ldots, v'_m$ をそれぞれ t の頂点とする（$\{v, v_1, \ldots, v_n, v'_1, \ldots, v'_m\} \subseteq V[t]$）．$t$ の局所変数は a, b, c の 3 つで，ソートはそれぞれ S_a, S_b, S_c であるとする．t の出力ソートを S_t とする．$events(t) = \{e_1, \ldots, e_\ell\}$ とする．各頂点 u に対し，$\mathrm{Name}(u)$ は u 自身，$\mathrm{Sname}(u)\,, \mathrm{Uname}(u)\,, \mathrm{Vsname}(u)\,, \mathrm{Vuname}(u)$ はそれぞれ $\mathrm{S\_}u\,, \mathrm{U\_}u\,, \mathrm{Vs\_}u\,, \mathrm{Vu\_}u$ で表す．

(1) v から各 v_i $(1 \le i \le n)$ への辺があり，v から出る辺はそれだけとする．v がユーザイベントで，v のデータ属性が「$a \in f(b) : S_a$」であるとする（図 2.9(a)）．このとき，

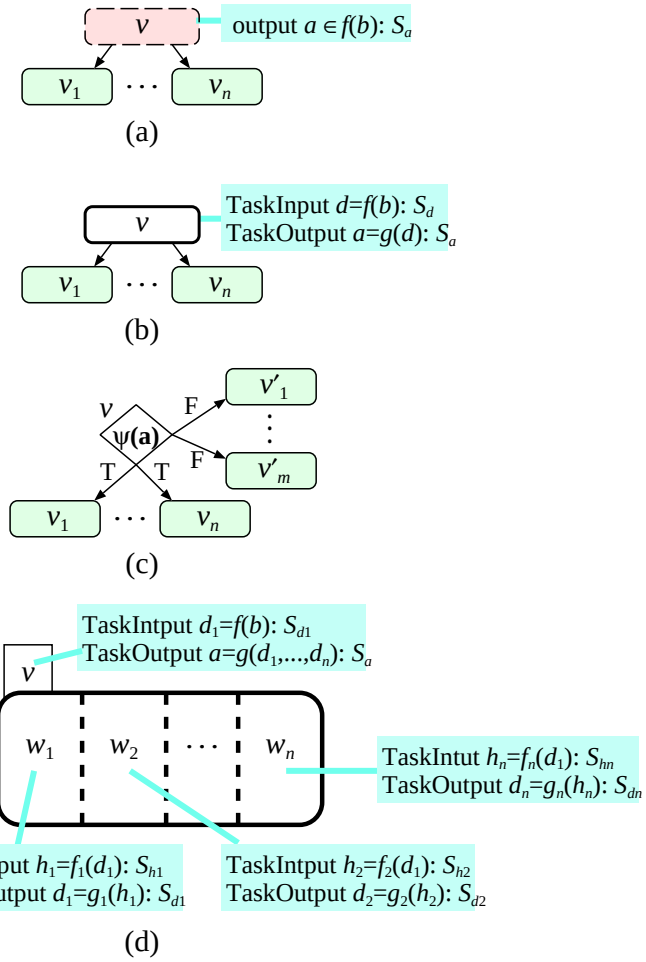


図 2.9　タスク図の一部

LOTOS 仕様中の v に関する部分（$\langle \mathrm{Vsproc}(t,v)\rangle$, $\langle \mathrm{Vuproc}(t,v)\rangle$ から導出される記号列）は図 2.10 のようになる．上述のように，変数宣言（“?” 変数 “:” ソート）で宣言される変数（ここでは a）は，プロセス Vs_v , Vu_v の仮引数とは異なる名前（a_1）に変えられている．つまりプロセス Vs_v , Vu_v 内では仮引数 a の値は無視され，イベント v の実行によって得られた a_1 の値が以降のプロセス Vs_v_i , Vu_v_i $(1 \leq i \leq n)$ に渡される．

(2) 上記 (1) と同様に辺があるとする．vが合成タスクで，v のデータ属性が「TaskInput $d = f(b) : S_d$ TaskOutput $a = g(d) : S_a$」であるとする（図 2.9(b)）．ただし d は v

```
process Vs_v [e_1, ..., e_ℓ] (a : S_a, b : S_b, c : S_c): exit(S_t) :=
  v ?a_1 : S_a; (Vs_v_1[e_1, ..., e_ℓ](a_1, b, c) [] ··· [] Vs_v_n[e_1, ..., e_ℓ](a_1, b, c))
endproc

process Vu_v [e_1, ..., e_ℓ] (a : S_a, b : S_b, c : S_c): exit(S_t) :=
  v ?a_1 : S_a [a_1 ∈ f(b)];
  (Vs_v_1[e_1, ..., e_ℓ](a_1, b, c) [] ··· [] Vs_v_n[e_1, ..., e_ℓ](a_1, b, c))
endproc
```

図 2.10 図 2.9(a) に対応する部分 LOTOS 仕様

```
process Vs_v [e_1, ..., e_ℓ] (a : S_a, b : S_b, c : S_c): exit(S_t) :=
  S_v[e_1, ..., e_ℓ'](f(b)) ⟫accept a_1 : S_a in
  (Vs_v_1[e_1, ..., e_ℓ](a_1, b, c) [] ··· [] Vs_v_n[e_1, ..., e_ℓ](a_1, b, c))
endproc
process S_v [e_1, ..., e_ℓ'] (d : S_d): exit(S_a) :=
...
```

図 2.11 図 2.9(b) に対応する部分 LOTOS 仕様

の仮引数，S_d は d のソートとする．また，$events(v) = \{e_1, \ldots, e_{\ell'}\}$ とする．このとき，LOTOS 仕様中の v に関する部分は図 2.11 のようになる（$\langle \mathrm{Vuproc}(t, v) \rangle$ から導出される記号列は $\langle \mathrm{Vsproc}(t, v) \rangle$ から導出されるものと同様であるので省略する）．この場合も (1) と同様だが，イベント v の実行の代わりにプロセス S_v の呼出しが行われている．プロセス S_v は，合成タスク v の内容を実行した後，exit(出力項 (v)) を実行して終了するプロセスである．Vs_v では S_v の出力を 出力変数 (v) すなわち a に受け取り，以降のプロセス Vs_v_i $(1 \leq i \leq n)$ のいずれかに制御を移す．ここでも (1) と同様に，変数 a の名前が a_1 に変えられている．

(3) v が条件分岐頂点で，v のデータ属性（条件）が「$\psi(a)$」であるとする．また，v から各 v_i $(1 \leq i \leq n)$ へはラベル T の付いた辺，各 v'_j $(1 \leq j \leq m)$ へはラベル F の付いた辺があり，v から出る辺はそれだけとする（図 2.9(c)）．このとき，LOTOS 仕様中の v に関する部分は図 2.12 のようになる．

(4) 上記 (2) において v が並行タスクであるとき，プロセス S_v , U_v がどのように定

process Vs_v [$e_1, \ldots, e_\ell$] ($a : S_a$, $b : S_b$, $c : S_c$): exit(S_t) :=

[$\psi(a)$] $\rightarrow$ (Vs_v_1[$e_1, \ldots, e_\ell$](a, b, c) [] $\cdots$ [] Vs_v_n[$e_1, \ldots, e_\ell$](a, b, c))

[] [not($\psi(a)$)] $\rightarrow$ (Vs_v'_1[$e_1, \ldots, e_\ell$](a, b, c) [] $\cdots$ [] Vs_v'_m[$e_1, \ldots, e_\ell$](a, b, c))

endproc

図 2.12 図 2.9(c) に対応する部分 LOTOS 仕様

process S_v [$e_1, \ldots, e_{\ell'}$] ($d_1 : S_{d_1}$): exit(S_a) :=

(S_w_1[$e_{11}, \ldots, e_{1\ell_1}$]($f_1(d_1)$)) $\gg$accept d_1_1 : S_{d_1} in exit(d_1_1, any S_{d_2}, ..., any S_{d_n}))

||| $\cdots$ |||

(S_w_n[$e_{n1}, \ldots, e_{n\ell_n}$]($f_n(d_1)$)) $\gg$accept d_n_1 : S_{d_n} in exit(any S_{d_1}, ..., any $S_{d_{n-1}}$, d_n_1))

$\gg$accept d_1_1: $S_{d_1}, \ldots, d_n$_1: S_{d_n} in exit($g(d_1$_1$, \ldots, d_n$_1$)$)

endproc

図 2.13 図 2.9(d) に対応する部分 LOTOS 仕様

義されるか述べる．$v = \{w_1, \ldots, w_n\}$ とする．v の局所変数は $d_1, \ldots, d_n$ であり，ソートはそれぞれ $S_{d_1}, \ldots, S_{d_n}$ であるとする．また，v のデータ属性は「TaskInput $d_1 = f(b) : S_{d_1}$ TaskOutput $a = g(d_1, \ldots, d_n) : S_a$」であるとする．各サブタスク w_i $(1 \leq i \leq n)$ は合成タスクで，w_i のデータ属性は「TaskInput $h_i = f_i(d_1) : S_{h_i}$ TaskOutput $d_i = g_i(h_i) : S_{d_i}$」であるとする（図 2.9(d)）．ただし h_i は w_i の仮引数で，S_{h_i} は h_i のソートとする．また，$events(w_i) = \{e_{i1}, \ldots, e_{i\ell_i}\}$ とする．このとき，LOTOS 仕様中の S_v の定義（$\langle$Sproc(v)$\rangle$ から導出される記号列）は図 2.13 のようになる（U_v の定義も同様）．図 2.13 からわかるように，プロセス S_v はプロセス S_w_1 , ..., S_w_n を非同期に並行実行するものとなる．この並列合成では各成分が S_w_i を実行した後，全成分が同期して exit を実行する．このとき，並列合成の第 i 成分（S_w_i を実行した成分）は，exit の第 i 引数に 出力変数 (w_i) を，第 j 成分 $(j \neq i)$ に「“any” 出力ソート (w_j)」を指定する．これにより，並列合成部分の終了後に実行される部分において，全 S_w_i の出力を得ることができる．プロセス S_v はこれらの出力（d_i $(1 \leq i \leq n)$）に基づいて 出力項 (v)（$g(d_1, \ldots, d_n)$）を出力し，終了する．

最後に，図 2.3 のタスクモデルの意味を表す LOTOS 仕様の一部を図 2.14 に示す．

```
specification Spec[···]: exit
behaviour
   S_通販で本を購入する [···] ‖ U_通販で本を購入する [···]
where
process S_通販で本を購入する [···]: exit :=
   ···
endproc
···
process S_キーワードで探す [···]: exit(Book) :=
   Vs1[···](⟨⟩ of String, {} of SetOfBook, B1 of Book)
endproc
process Vs1[···](k: String, blist: SetOfBook, b: Book): exit(Book) :=
   Vs_キーワードを指定する [···](k, blist, b)
endproc
process Vs_キーワードを指定する [···](k: String, blist: SetOfBook,b: Book): exit(Book) :=
   キーワードを指定する?k_1:String; Vs2[···](k_1, blist, b)
endproc
process Vs2[···](k: String, blist: SetOfBook, b: Book): exit(Book) :=
   let blist_1 = Search(k): SetOfBook in Vs3[···](k, blist_1, b)
endproc
process Vs3[···](k: String, blist: SetOfBook, b: Book): exit(Book) :=
   [not(blist = ∅)] –⟩ Vs_キーワードにマッチする本の一覧を表示する [···](k, blist, b)
   [] [blist = ∅] –⟩ Vs_見つからなかった旨表示する [···](k, blist, b)
endproc
process Vs_キーワードにマッチする本の一覧を表示する [···](k: String, blist: SetOfBook, b:
Book): exit(Book) :=
   キーワードにマッチする本の一覧を表示する!blist;
   (Vs_一覧から本を選ぶ [···](k, blist, b) [] Vs_戻る [···](k, blist, b))
endproc
process Vs_見つからなかった旨表示する [···](k: String, blist: SetOfBook, b: Book): exit(Book)
:=
   見つからなかった旨表示する; Vs_キーワードを指定する [···](k, blist, b)
endproc
process Vs_一覧から本を選ぶ [···](k: String, blist: SetOfBook, b: Book): exit(Book) :=
   一覧から本を選ぶ?b_1:Book; exit(b_1)
endproc
process Vs_戻る [···](k: String, blist: SetOfBook, b: Book): exit(Book) :=
   戻る; Vs_キーワードを指定する [···](k, blist, b)
endproc
···
```

図 2.14 図 2.3 の意味を表す LOTOS 仕様（一部）

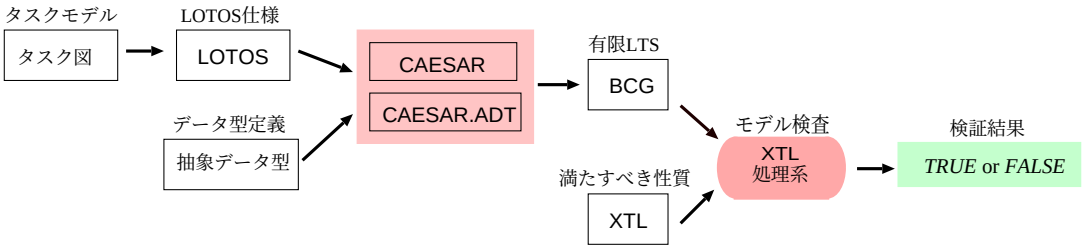


図 2.15　検証手順

2.6　タスクモデルの形式的検証

2.6.1　検証問題の定義

作成されたタスクモデルにおいて，システムが満たすべき性質が成り立つかどうかを検証する．2.5.1 節で述べたように，タスク図の意味は LOTOS 仕様の形で表現される．タスクモデルの意味を表す LOTOS 仕様 (正確にはその LOTOS 仕様が表す LTS）を検証対象システムのモデルとする．

システムが満たすべき性質は，CTL(Computation Tree Logic) や様相 μ 計算 [24] などの時相論理式を用いて記述する．本論文では CTL よりも表現能力の高い様相 μ 計算を用いることにする[1]．

本研究で扱う検証問題とは，タスクモデル tm と検証したい性質 ϕ が与えられたとき，tm のもとで ϕ が成り立つか，すなわち，tm の意味を表す LTS のもとで ϕ が成り立つか，を調べることである．以下，2.6.2 節で検証手順を述べ，実際に検証を行った例を 2.6.3 節で述べる．

2.6.2　検証手順

LOTOS 仕様が様相 μ 計算で書かれた論理式を満たすかどうかをモデル検査手法 [24] を用いて検証する．そのような検査ツールとして CADP[25]，CWB-NC[26, 27] などがある．ただし，CWB-NC ではシステム記述を基本 LOTOS で与えるため，タスクモデル上のデータ属性を直接扱うことができない．したがって，ここでは CADP を用いることにする．

図 2.15 に検証手順を示す．まず，CAESAR および CAESAR.ADT(CADP ツール群に

[1] ここでは，読みやすさのため CTL の演算子を交えて論理式を書くが，様相 μ 計算の演算子でも記述できる．

含まれる）を用いて，タスクモデルの意味である LOTOS 仕様を有限な LTS に変換する．整数型や文字列型など，無限の要素を持つソートを使用している場合，LOTOS 仕様が表す LTS は本来無限の大きさとなるが，CADP ではこれらのソートの定義域を別途指定させる（例えば，整数型の値は 0 から 4 までの 5 種類とする，など）ことで，有限の LTS を得る．検証したい性質は，CADP において定義される XTL という言語の構文にしたがって，様相 μ 計算の形で記述する．最後に，これらの LTS と論理式を XTL 処理系に入力しモデル検査を行う．検証結果として $true$ あるいは $false$ を得る．

2.6.3 検証例

ここでは，図 2.3 のタスクモデル「通信販売で本を購入する」に対する検証例を示す．このタスクモデルが満たすべき性質として以下のようなものが考えられる．

性質 1 「何回でも本をキーワード検索できる」

$\nu Z.\mathsf{EF}\langle$ キーワードを指定する $\rangle\langle$ キーワードにマッチする本の一覧を表示する $\rangle Z$

性質 2 「本をキーワードで探した後，分野で探すこともできる」

$\mathsf{AG}([$キーワードを指定する$](\mathsf{EF}\langle$ 分野を指定する $\rangle true))$

性質 3 「本を購入できる」

$\mathsf{EF}\langle$ 確認してから発注 $\rangle true$

性質 4 「システムは必ず終了する」

$\nu Z.\mathsf{EF}\langle \mathrm{exit} \rangle true \ \wedge \ [\neg \mathrm{exit}]Z$

性質 5 「本の検索結果（一覧）が表示されたときは必ず一覧の中から本を指定できる」

$\mathsf{AG}([$キーワードにマッチする本の一覧を表示する$]\langle$ 一覧から本を選ぶ $\rangle true)$

以上の各性質に対して CADP を使って検証を行った結果を表 2.2 に示す．実験には Sun SPARCstation 20(主記憶 96MB) を用いた．またこのタスクモデルから得られた LTS の状態数は 3,373，状態遷移数は 13,610 であった．ただし，整数型の値は 0 から 4 までの 5 種類，文字型の値は空文字列を含めて 4 種類として LTS への変換を行った．

表 2.2 からわかるように，このタスクモデルは性質 2 を満たさない．これを満たすように修正するには，タスク「キーワードで探す」を終了した後，再び任意の検索方法で本を検索できるようにすればよい．これは，図 2.17 の空頂点 (b) から空頂点 (a) への遷移を追加すれば実現できる．

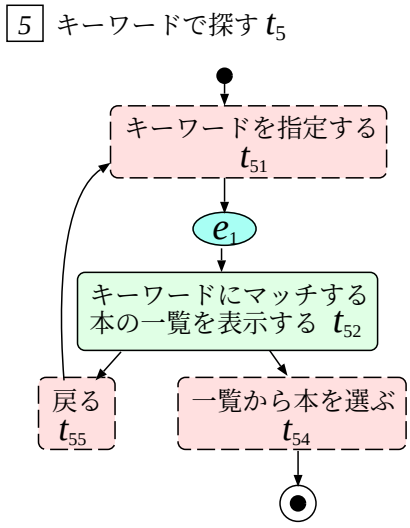


図 2.16 条件分岐頂点の欠落した例

また，このタスクモデルは性質 5 を満たしているが，タスク「キーワードで探す」を図 2.16 のような単純な形で与えた場合には性質 5 を満たさない．これは，空頂点のデータ属性 $e_1.blist = Search(t_{51}.k)$ において $e_1.blist = \emptyset$ となる可能性があるからである．$e_1.blist = \emptyset$ のとき $t_{54}.b \in t_{54}.blist$ となる $e_1.b$ は存在しないため，イベント「一覧から本を選ぶ」を実行できない．このように，一見ある性質を満たすようでも，全ての実行系列がその性質を満たすとは限らないことはしばしばあり，人間の直観に頼って仕様の正当性を確認することは，結果の信頼性の点で問題がある．今回は小規模なタスクモデルを例としたが，仕様が大規模化・複雑化するにつれ，確認時に思い込み・見落し等のヒューマンエラーが発生する可能性はより高くなる．形式的検証を行なえば，このようなヒューマンエラーの影響を受けず，信頼性の高い結果を得ることができる．

このように，形式的検証を行うことで，プロトタイプの構築以前にシステムの問題を検出することができ，システム開発の効率を改善することができる．

2.7 UI 設計情報に基づくプロトタイプの自動生成

プロトタイプの実装を可能とするため，データ属性に基づいた UI 設計情報をタスクモデルに記述する．UI 設計情報とは，タスクモデルの各タスクに割り当てるウィンドウや，

表 2.2 検証結果

満たすべき性質	検証結果	検証にかかった時間 (s)
性質 1	TRUE	1
性質 2	FALSE	6
性質 3	TRUE	9
性質 4	TRUE	19
性質 5	TRUE	1

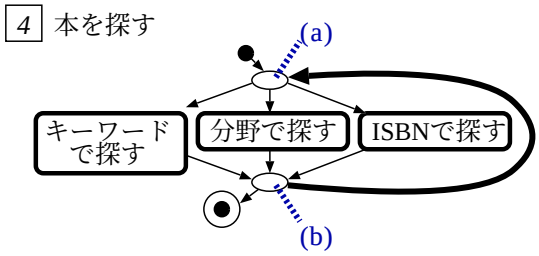


図 2.17 タスクモデルの更新

データ属性に書かれた各データの送受信のために利用される UI 部品に関する記述である．UI 設計情報を追加したタスクモデルからプロトタイプを自動生成する．UI 設計情報は実装の枠組に依存する．実装の枠組とは，使用可能な UI 部品型の集合や，プロトタイプを記述するプログラミング言語などを規定するものであり，HTML, Visual Basic, Java などが実装の枠組として考えられる．本研究では，UI 部品型の集合が規定されており，プログラミング言語に自由度のある WWW を実装の枠組とした場合を中心に説明する．またここでは，プロトタイプを記述するプログラミング言語として Java を採用している．

2.7.1 UI 設計情報の記述

システムとユーザの間の送受信は，入力欄やボタンなどの **UI 部品**を通じて行われる．そこで，各イベントについて，それが使用する UI 部品を割り当てる．UI 部品の介在により，ユーザとシステムが共に参照できる領域にデータが格納される（図 2.18）．また，UI 部品を表示するための画面の一領域を**フレーム**と呼ぶことにし，タスクにフレームを静的に割り当てる．

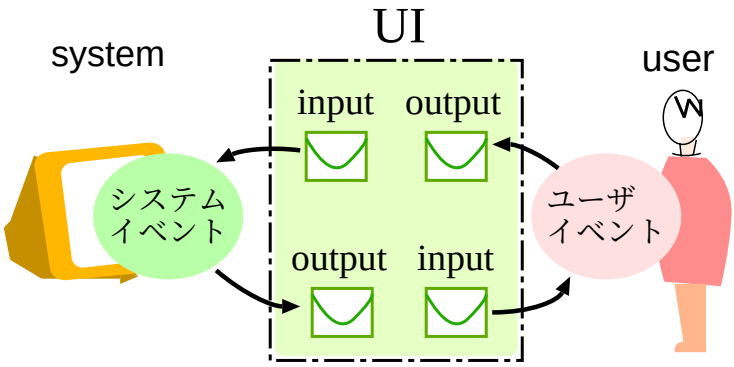


図 2.18　イベント間のデータの共有

UI 部品の指定

各タスクの各データに対して，タスク t が使う UI 部品を次のように指定する．

ui UI 部品名 : [データ] : UI 部品型

ここで，データはその UI 部品が作用するデータを表す．各データ属性に対する UI 部品の割り当てを次のように行う．

a. システムイベントの各 output データには，システムがデータを表示するための UI 部品 (e.g., document, etc.) を割り当てる．

b. ユーザイベントの各 output データには，ユーザがデータの入力を行うための UI 部品 (e.g., 1linetext, textarea, etc.) を割り当てる．

c. b. の UI 部品が存在する場合，その UI 部品に入力されたデータをシステムに通知するための UI 部品 (e.g., submit) を割り当てる．そうでなければ，次のタスクに遷移するための UI 部品 (e.g., link) を割り当てる．

図 2.19 はタスクモデルに UI 設計情報を記述した図である．ユーザイベント「キーワードを指定する」には，UI 部品型が 1linetext である UI 部品 x_1 とデータ $t_{51}.k$ が指定されている．これは，図 2.7 でのデータ属性の記述 output : $t_{51}.k$: String に対して割り当てられた UI 部品である．また submit 型の UI 部品 s_1 は，UI 部品 x_1 に入力されたデータ $t_{51}.k$ をシステムに通知するための UI 部品である．

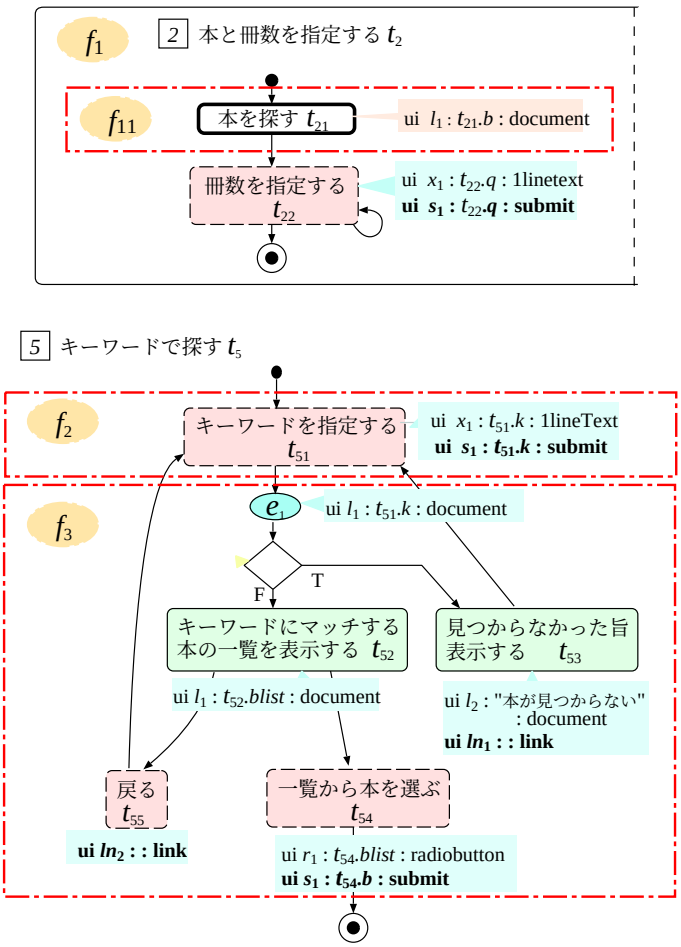


図 2.19　UI 設計情報の記述

UI 部品型

UI 部品型には，単純 UI 部品型と，単純 UI 部品型 U の要素の集合を表す SetOf$\langle U \rangle$ 型がある．

単純 UI 部品型　使用可能な単純 UI 部品型の集合は，対象とする実装のアーキテクチャによって決まる．すなわち，使用可能な UI 部品型の集合や，プロトタイプを記述するプログラミング言語などに依存する．例えば，WWW を実装のアーキテクチャとした場合，単純 UI 部品型として以下のようなものが考えられる．

- 1linetext …一行のテキスト入力欄

表 2.3 WWW における UI 部品

	SetOf	システムへの通知	入力の保持	属性 form の指定
1linetext	○		○	必要
textarea	○		○	必要
radiobutton	○		○	必要
checkbox	○		○	必要
submit		○		必要
link	○	○		
document	○			

- textarea …複数行のテキスト入力欄
- radiobutton …選択ボタン（複数選択が不可能）
- checkbox …選択ボタン（複数選択が可能）
- submit …上記の各部品に入力した値をシステムに通知するボタン
- link …ハイパーリンク（選択がただちにシステムに通知される）
- document …出力欄

SetOf$\langle U \rangle$ 型　SetOf$\langle U \rangle$ 型は, 定義域となる型 U を引数として定義される複合型である. 定義域となることができる UI 部品型は限られている (表 2.3).

システムへの通知　UI 部品は, その表示中, システム本体と独立に動作し, ユーザの操作に応じることができる. 一部の UI 部品は, 受信待ち状態にあるシステムに対し, ユーザの操作があったことを通知する. これによりシステムは受信待ち状態を脱する. このシステムに通知を行う UI 部品は各ユーザイベントに対してちょうど一個含まれる. 上で挙げた UI 部品のうち WWW においてシステムへの通知を行う部品は，submit と link である（表 2.3）. また, U がシステムへの通知を行う型の場合, SetOf 型 $S\langle U \rangle$ のオブジェクトも通知を行う.

ユーザからの入力の保持　一部の UI 部品型の UI 部品は, ユーザが入力した値を内部に保持する. これを読み出すメンバ関数を用いて, システムはユーザが入力した値を得る. WWW においてユーザの入力を内部に保持する UI 部品型は, 1linetext，textarea,

radiobutton，checkbox である（表 2.3）．また, U が入力を保持する型の場合, SetOf 型 $S\langle U\rangle$ のオブジェクトも入力を保持する．

メンバ関数　各 UI 部品型は独自のメンバ関数を持つ．

各単純 UI 部品型は以下のメンバ関数を持つ．

- show() ： 自身を表示する．
- entered() ： ユーザが入力した値を返す (値を保持する型のみ)．

SetOf 型 $S\langle U\rangle$ は以下のような機能のメンバ関数を持つ．

- 要素のうちユーザが選択したものの番号を返す関数
- 第 i 番の要素を返す関数

フレームの割り当て

任意のタスクに任意のフレームを割り当てることができる．フレームが切り替わる部分には，次のフレームの表示開始を指示するための UI 部品が自動的に挿入される．

図 2.19 では，タスク「本と冊数を指定する」にはフレーム f_1 が割り当てられており，そのサブタスク「本を探す」にはフレーム f_{11} が割り当てられている．したがって，フレーム f_1 には，フレーム f_{11} の表示の開始を指示するための UI 部品が挿入される．図 2.20 のフレーム「本と冊数を指定する」は，フレーム「本を探す」の表示の開始を指示するための UI 部品が挿入されており，これをクリックするとフレーム「本を探す」が表示される．

ただし，1 フレームに対応するイベントの集合は，1 つ以上のユーザイベントを含み，イベントの実行順の最後は，システムへの通知を行なう UI 部品を含むユーザイベントになるように指定しなければならない．

図 2.19 において，フレーム f_3 が割り当てられているイベントの集合には，submit を含むユーザイベント「一覧から本を選ぶ」や link を含む「戻る」が存在し，それらが実行順の最後であるように指定している．

2.7.2　プロトタイプの自動生成

タスクモデルに基づいてプロトタイプを自動生成する．すなわち，フレームおよび UI 部品の表示や，システムとユーザ間の入出力を制御するプログラムを生成する．

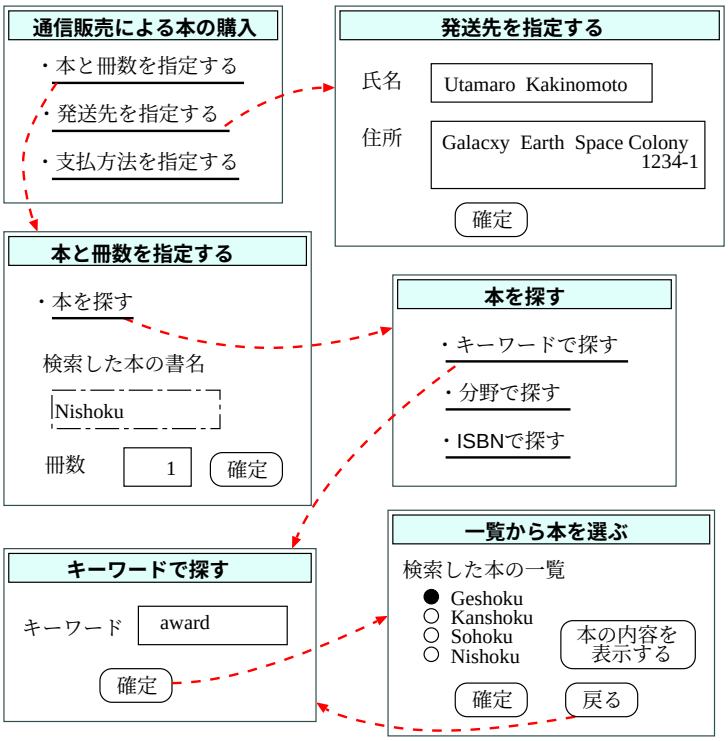


図 2.20 プロトタイプのユーザインタフェース

プロトタイプ生成の基本方針として，1 フレームに対してプロトタイプの 1 モジュールを生成し，モジュール内は各イベントに対応するプログラムコードを並べることで構成する．モジュール間のデータの授受はグローバル変数を介して行う．

プロトタイプへの変換アルゴリズムは，対象とする実装のアーキテクチャごとに与えられる．以下では例として，WWW を実装のアーキテクチャとしたときの変換手順を説明する．

WWW アーキテクチャでのプロトタイプ生成

各フレーム f に対して，以下のような 2 つの CGI プログラムを生成する．

- $P_{f,1}$: 必要な内部処理の後，フレームおよび UI 部品を表示する．
- $P_{f,2}$: ユーザの操作によって起動され，ユーザの入力を変数に代入するなどの処理をする．

各 CGI プログラムが共有すべきグローバル変数は，ファイルを媒体に使って実装する．

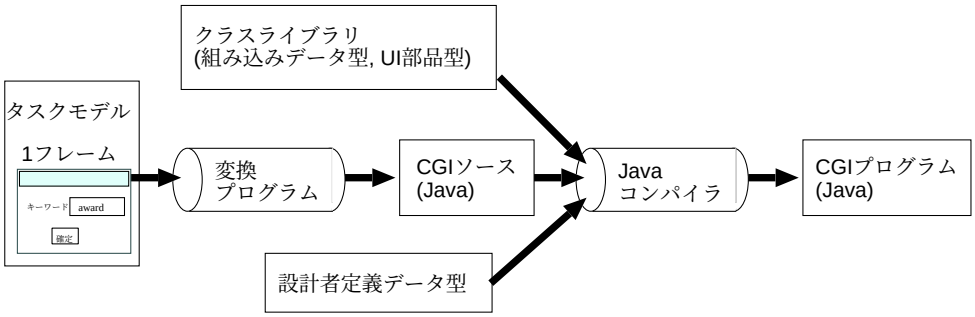


図 2.21 プロトタイプ生成システム

$P_{f,1}$ の生成アルゴリズムは，タスクモデル上で，フレームの入口に当たる各イベントから，システムイベントのみを辿っていける範囲で深さ優先探索を行い，各イベントに応じたプログラムコードを生成していく．また，最後にフレームおよびフレーム中の全 UI 部品を表示するコードを挿入する．$P_{f,2}$ の生成アルゴリズムは，各ユーザイベントから探索を行い，各イベントに応じたコードを生成していく．

各イベントに応じたプログラムコードは以下のようになる．

- システムイベント: データベース参照などの内部処理および output データの表示
- ユーザイベント: output データ（UI 部品に対する入力値）の読み出し

試作したプロトタイプ自動生成システムの概要

このアルゴリズムに基づき，タスクモデルからプロトタイプを自動生成するシステムを試作した (図 2.21)．

プロトタイプの実装の枠組は WWW とし，CGI プログラムおよびデータ型，UI 部品型を記述する言語は Java としてシステムを作成した．このシステムは以下のもので構成される．

- 変換プログラム
  タスクモデルを入力として, プロトタイプのコードを生成し出力するプログラム
- クラスライブラリ
  組み込みデータ型，UI 部品型の定義
- 設計者定義データ型の定義

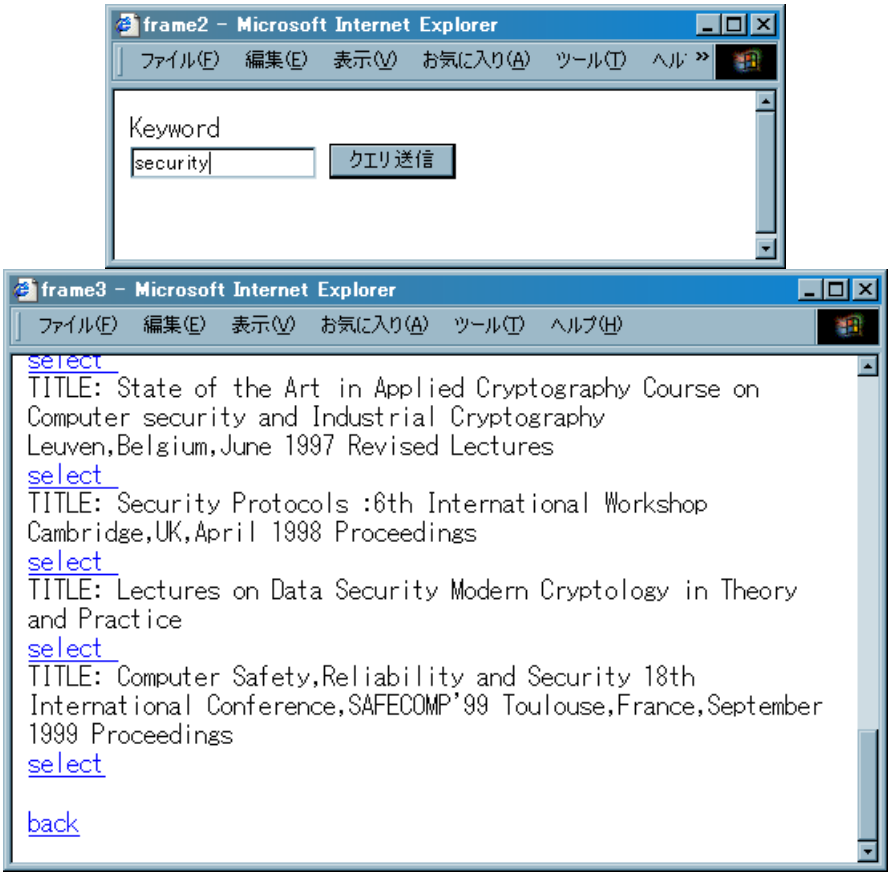


図 2.22 プロトタイプの実行例

タスクモデルをテキスト表現したものを変換プログラムへの入力とする．変換プログラムは，その入力を CGI ソースプログラムに変換する．そして, この CGI ソースプログラムと, クラスライブラリ (組み込みデータ型, UI 部品型)，設計者定義データ型を合わせてコンパイルすることで，プロトタイプが完成する．

図 2.22 は，図 2.19 のタスクモデルにおけるフレーム f_2，f_3 から生成されるプロトタイプの実行時の画面である．

2.7.3 プロトタイプの評価

プロトタイプを用いてユーザおよび開発者によるシステムの評価を行ない，その評価にしたがってタスクモデルの更新を行う．この評価においては，UI 部品の配置，色，大

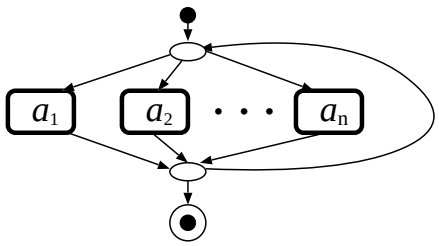


図 2.23 タスクモデルの部分構造の一例

きさや，表示される文言などが適切かどうかが判断される．フレームの遷移の発生箇所が不自然な場合や， 視覚的要素や文言における不具合を解決するために 1 フレームに表示する UI 部品の個数を変更する場合，UI 設計情報の更新が必要となる．その際は，タスクモデル中の該当する部分においてデータの依存関係を考慮しながら，1 フレームに割り当てるタスクを変更する（フレームの遷移箇所を変更する）．

2.8 第 2 章のまとめ

本章では，アプリケーション開発工程の上流から適用できる体系化されたインタラクティブシステムの設計法を提案した．

まず，ユーザタスクを図的言語 タスク図 を用いてタスクモデルとして記述する方法について示した．そして，そのタスクモデルから LOTOS 仕様への変換規則を与えることにより形式的意味定義を行った．次に，モデル検査法を用いて，例題 「通信販売による本の購入」に対して形式的検証を行い，その有効性を確認した．また，タスクモデルのデータ属性に基づき UI 設計情報を追加し，プロトタイプの自動生成を行なう方法を示した．

今後の課題としてタスク図の拡張，たとえば，

- タスクの中断や割り込み／復帰などの記述
- タスクの実行時間やイベントの発生時刻に関する制約の記述

などを行なえるように拡張することが考えられる．また，これらの機能の意味定義には時間拡張 LOTOS[28] などが有効であると考えられる．また，シグナル送受信の記述に関しても考慮すべきと考える．

本設計法に基づく支援環境の実現も重要課題である．その際，現在作成済みのプロトタイプ自動生成機能 [29] の他，タスク図の記述と詳細化を支援するための編集機能，タスクモデルから LOTOS 仕様への自動変換機能を実装することが必要と思われる．

また，形式的検証において，どのような性質を検証すべきかを決定し，様相 μ 計算などの論理式に書き下す作業は熟練を要する．検証すべき性質の記述を支援する機能として以下が考えられる．

- 典型的な性質を表す論理式のテンプレートを提供する機能
- タスクの部分構造から検証すべき性質を示唆する機能

例えば，図 2.23 のように，ユーザがタスクの選択を行うタスクモデルの場合，選択されるタスクを $a_1, a_2, \ldots, a_n$ とすると，

$$\mathsf{AG}([a_1](\mathsf{EF}\langle a_2\rangle true)) \ \wedge \ \mathsf{AG}([a_2](\mathsf{EF}\langle a_3\rangle true)) \ \wedge \ \ldots \wedge \mathsf{AG}([a_n](\mathsf{EF}\langle a_1\rangle true))$$

という式を自動的に導出できれば論理式を記述する負担を軽減できる．このようなテンプレートを用意するなどの設計支援法についても今後考察したい．

第3章　ユーザインタフェースの代数的仕様記述とその実現

3.1　はじめに

本章では, 代表的な形式的記述法の一つである代数的仕様記述法 [7] を用いた UI 設計法を提案する. 代数的仕様記述法の部分クラスとして, 抽象的順序機械型代数的仕様 (ASM 仕様) がある [8]. まず，UI の仕様を ASM 仕様として記述するために，多プロセスモデルである抽象的ウィンドウシステム（AWS）を導入する (3.2 節). AWS は，UI モジュール間の通信をメッセージとメッセージ処理用キューを用いて表現したモデルである．3.3 節では，AWS にしたがって ASM 仕様記述を行なう方法とその適用例を示す．適用例として，1 加算するボタンや 1 減算するボタンを用いてウィンドウに表示されている数値を変更する機能を持つ UI を例に本手法を説明する．まず，ASM 仕様を大きく 4 つの部分で構成し，各部分を記述する方法について説明を行なう．設計者はその内の一つの部分である “設計対象 UI の仕様” を記述し，予め用意された残り 3 つの部分 “組み込み UI 部品”，“AWS の機能部分”，および “基本データ型” に関する記述と組み合わせて AWS 仕様を完成させる．3.4 節では，ASM 仕様から Java プログラムへのコンパイラについて述べる. 試作したコンパイラを用いて, 3.3 節で示した仕様記述例をコンパイルした結果について述べる. また，ウィンドウ遷移機能を持つシステム，キーワードを用いて本を検索するシステムの一部に本手法を適用した例を紹介する.

3.2　抽象的ウィンドウシステム

モジュール単位での仕様作成や再利用を容易にするため，本仕様記述法では抽象的ウィンドウシステム（AWS）というモデルを導入する．AWS は，一般的なオブジェクト指向ツールキット X toolkit[30], Swing[31] などで構成される GUI アプリケーションを単純化したものであり，個々の UI モジュールがメッセージ送受信によって非同期動作するよう

図 3.1 SetValue の画面表示例

な多プロセスモデルとなっている．AWS 上の各 UI モジュールを**オブジェクト**と呼ぶ．

以下のような機能をもつ簡単な UI モジュール（SetValue[16]）を例に説明する（図 3.1）．この UI は, 以下のような機能をもつ.

- 内部にカウンタを 1 つもち, 4 つのボタン up, down, reset, ok と説明文，現在のカウンタ値を表示する.
- ウィンドウ起動時，カウンタの初期値を表示する．
- ボタン up をクリックすると，カウンタに 1 加算した値を表示する．
- ボタン down をクリックすると，ウンタに 1 減算した値を表示する．
- ボタン reset をクリックすると，カウンタ値は初期値にリセットされる．
- ボタン ok のクリックにより, この UI を生成した親オブジェクトのコールバックメソッドを呼び出す．このとき, その引数としてカウンタ値が親オブジェクトに渡される．そして，各 UI 部品とともにウィンドウが消滅し処理が終了する．

図 3.2 は図 3.1 の AWS における各オブジェクトの構成である．ここでは SetValue オブジェクトを再利用可能なモジュールとする目的で，SetValue オブジェクトの起動などを行うオブジェクト（Example オブジェクト）を SetValue オブジェクトとは別に置く．Example オブジェクトは，SetValue を UI として利用するアプリケーション本体の一例である．SetValue オブジェクトは，2 個の Label オブジェクトと 4 個の Button オブジェクトを構成分子として含んでいる．AWS 上では，これらの構成分子も独立したオブジェクトとして扱われる．すなわち，各 Button オブジェクトや Label オブジェクトは直接 AWS の管理下にある．

SetValue オブジェクト自身は，カウンタ値を管理する機能と，自分の構成分子であるオブジェクトと通信する機能だけを持つ．これらのオブジェクトが，AWS を通信媒体としてメッセージ送受信を行うことで，この UI の機能が実現される．ユーザの操作を表す

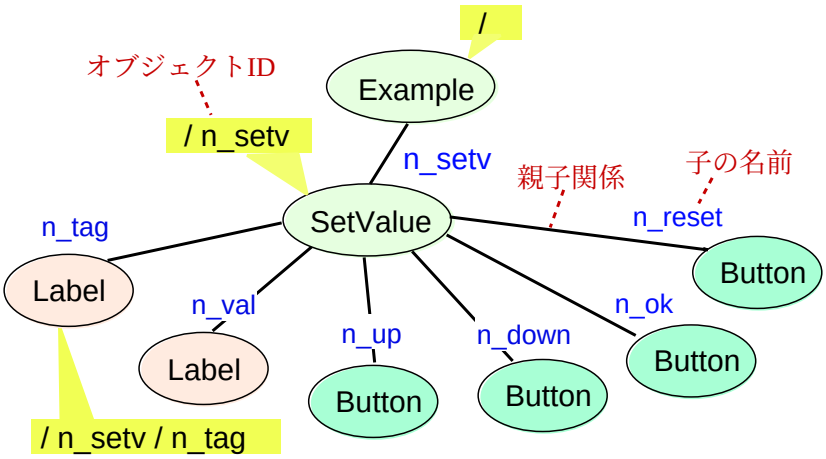


図 3.2　AWS における UI オブジェクトの構成例

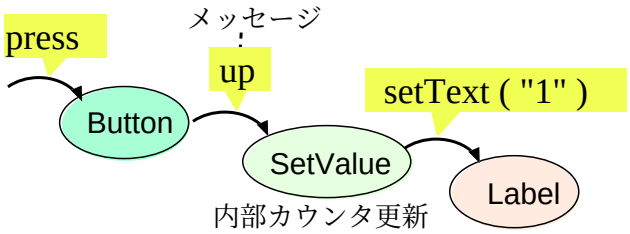


図 3.3　UI モジュール間通信

イベント，および，オブジェクト間の通信は，AWS のメッセージ配送機能を介して行われる．これを，SetValue 上の up ボタンがクリックされた場合を例に説明する（図 3.3）．up ボタンの押下は，AWS 外から Button オブジェクトへの press メッセージの送信として表される．AWS がこれを Button オブジェクトに配送する．press メッセージを受信した Button オブジェクトは，予め指定されたコールバックメッセージを SetValue オブジェクトに送信する．コールバックメッセージの受信により，SetValue オブジェクトは up ボタンが押されたことを知り，内部カウンタ値を増加させる．また，カウンタ値を表示している Label オブジェクトに，表示内容の更新を要求するメッセージを送信する．以降ではメッセージ送受信のことをメソッド呼出しと呼ぶことがある．このようにして，up ボタンが押されたときの機能が実現される．Label オブジェクトに表示内容を更新させるメッセージのように，メッセージには送信先に渡したいデータを引数として与えることができる．コールバックメッセージの指定方法は後で述べる．

AWS は，各オブジェクトおよびメッセージ処理用キューから構成され（図 3.4），以

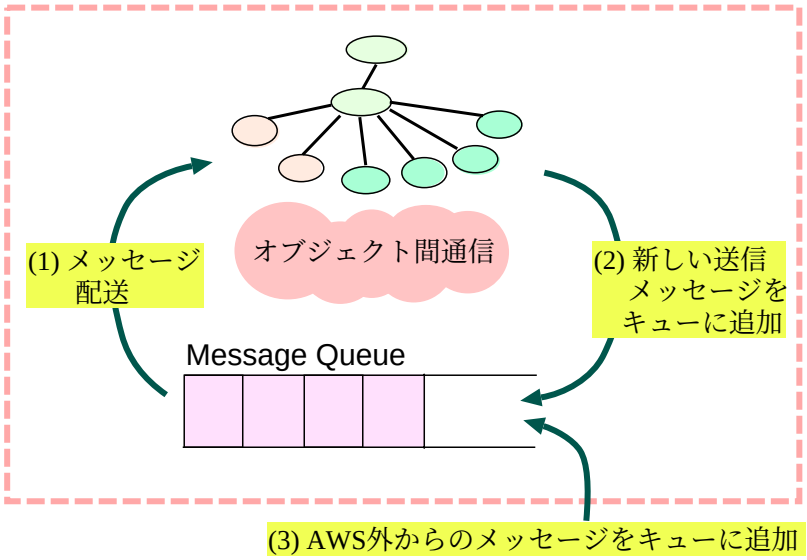


図 3.4 抽象的ウィンドウシステム

下の機能を持つ.

(1) メッセージ配送
キュー内の 1 メッセージを取り出し，送信先オブジェクトに配送する.

(2) 新しい送信メッセージをキューに追加
受信オブジェクトが新たに送信するメッセージをキューに追加する.

(3) AWS 外からのメッセージをキューに追加

AWS の下では，各オブジェクトの仕様は次から成る.

(1) そのオブジェクトが受け付けるメッセージ（メソッドとも呼ぶ）の集合

(2) 各メッセージの受信（メソッド呼出しとも呼ぶ）の際に新たに送信するメッセージ（一般に複数），および，自身の状態変化の内容

(1) には新しいオブジェクト（その仕様のインスタンス）を生成するメッセージ（生成メッセージ）が含まれる．生成メッセージが送信された場合，AWS は新しいオブジェクトを生成する．生成メッセージの受信（生成されたオブジェクトが行う）に対しても (2) を記述でき，他の生成メッセージを送信するよう記述されていた場合は連鎖的にオブジェクトが生成される．SetValue オブジェクトのように他のオブジェクトを構成分子として持つものは，通常このように，自身の生成時に構成分子を生成することとなる.

メッセージの送信先の指定方法について説明する．新しいオブジェクトの生成は，その仕様で定義された生成メッセージを AWS が処理することで行われ，オブジェクト o が生成メッセージを送信してオブジェクト o' を生成したとき，o を o' の親，o' を o の子と呼ぶ．メッセージ名自身を値として保持したり，送信メッセージの引数として受け渡したりすることが可能である．この機能を使って，子オブジェクトにコールバックメッセージを指定することができる．例えば，SetValue オブジェクトは up ボタンオブジェクトの生成時に，コールバックメッセージとして up メッセージを指定しておくよう記述されている．メッセージの送信先オブジェクトを指定するために，親オブジェクトは子の生成時に名前を指定し，以降，子にメッセージを送信する際はその名前を送信先として指定する．この名前は兄弟（同じ親に対する子）間で一意であればよい．一方，オブジェクトが親，あるいは，自分自身にメッセージを送信する際は，それぞれ特別な名前 “**parent**”，“**self**” を送信先として指定する．これにより，各オブジェクトは自分が子に付けた名前のみ使って送信先を指定できる．オブジェクト間の通信は親子間でのみ行える．もし親または子以外のオブジェクトにメッセージを送信したい場合は，自分から相手までの，親子関係を辺とするパス上の各オブジェクトにメッセージを中継してもらう必要がある．AWS は，オブジェクト間の親子関係と，親が子に付けた名前とを組み合わせて，階層型ディレクトリにおけるパス名と同様な識別子を使って各オブジェクトを一意に識別する（図 3.2）．この識別子を**オブジェクト ID** と呼ぶ．AWS 外からのメッセージ（例えば，3.4(3) でのユーザの操作を表すイベントなど）では，直接オブジェクト ID が送信先として指定される．

3.3 UI の代数的仕様記述

3.3.1 代数的仕様

代数的仕様とは, 3 項組 (S, F, AX) である．ここで,

- S はソートの有限集合
- F は S-シグニチャ, すなわち, S に属するソートの上の関数記号の宣言の有限集合 F に属する関数記号とソート付き変数の集合 X から構成されるソート s の項全体の集合を $T(F,X)_s$ と書き, 項全体の集合を $T(F,X) = \bigcup_{s \in S} T(F,X)_s$ と書く．ϕ で空集合を表す．変数を含まないソート s の項の集合を $T(F)_s = T(F,\phi)_s$, 変数を

含まない項の集合を $T(F) = T(F, \phi)$ と書く.

- AX は公理の有限集合ここで公理とは, あるソート $s \in S$ の項の順序対 $l = r$ である ($l, r \in T(F, X)$). l をこの公理の左辺, r を右辺と呼ぶ. s を強調して, ソート s の公理と呼ぶこともある.

以下の例および実装したコンパイラへの入力では, 仕様記述言語 LOTOS[6] 中で用いられている，抽象データ型定義言語 ACT-ONE[32] の構文に準拠して仕様を記述する (図 3.9–3.11 参照).

- S に属するソートを, キーワード **sorts** に続けて記述する.
- キーワード **opns** に続けて $f : s_1, s_2, \cdots, s_n \rightarrow s$ と記述することにより, 「引数のソートが $s_1, s_2, \cdots, s_n$, 関数値のソートが s である関数記号」f が F に含まれることを表す.
- キーワード **eqns** に続けて **ofsort** s $l = r$ と記述することにより, ソート s の公理 $l = r$ が AX に含まれることを表す. なお, 公理を記述するのに用いる変数 x とそのソート s を, **for all** $x : s$ のように記述する.

$SP = (S, F, AX)$ を代数的仕様とする. F から生成される項の集合上の AX のすべての公理を満たす $T(F)$ 上の最小の合同関係（厳密には, $T(F)_s$ 上の最小の合同関係の族）を, SP の指定する合同関係といい, $\equiv_{SP}$ で表す.

3.3.2 抽象的順序機械型代数的仕様

代数的仕様 $SP = (S, F, AX)$ が次の条件をすべて満たすとき，SP を ASM 仕様と呼ぶ.

(1) S は，抽象的状態を表すソート state を含む.

(2) F の関数は，次のように分類される.

(2.a) 状態遷移関数：順序機械の状態を遷移させる関数. state 型の引数を高々1 個持つ. 関数値のソートは state である.

(2.b) 状態成分関数：各状態における状態成分を表す関数. state 型の引数をちょうど 1 個持つ. 関数値のソートは state 以外である.

(2.c) 補助関数：引数のソート，関数のソートのどれも state 以外の関数.

(3) AX に現れる公理は次の条件を満たす.

(3.a) 左辺は線形で，かつ重なりを持たない [33].

(3.b) 右辺に表れる変数は，その左辺にも現れる.

(3.c) 各関数の意味を定義する公理は以下の形式である．ここで，T, T' を状態遷移関数，O を状態成分関数，A を補助関数とする．さらに，s を state 型の変数とし，$x_1, x_2, \ldots, x_n$，$u_1, u_2, \ldots, u_m$ を state 型以外の変数とする.

形式 1　$O(T(s, u_1, \ldots, u_m), x_1, x_2, \ldots, x_n) = \cdots$

形式 2　$T'(s, x_1, x_2, \ldots, x_n) = \cdots$

形式 3　$A(x_1, x_2, \ldots, x_n) = \cdots$

形式 1 および形式 3 の右辺に状態遷移関数が現れてはならない．したがって，状態遷移後の状態成分関数の値は，遷移前の状態における状態成分関数や補助関数の値と，状態遷移関数の state 以外のソートの引数のみに依存して定まる.

3.3.3　抽象的ウィンドウシステムの代数的仕様

抽象的ウィンドウシステム（AWS）の代数的仕様は，次の 4 つの部分から構成される（図 3.5）.

(1) 整数（Integer），文字列（String）などの基本データ型の仕様.
UI 設計に限らず一般に用いられる基本的なデータ型の仕様を与える．仕様記述言語 LOTOS[6] の標準データ型ライブラリなどで提供される仕様の部分集合となっている.

(2) 3.2 節で述べた AWS の各機能を実現する部分（BasicSystem と呼ぶ）の仕様.
メッセージを表すデータ型やメッセージ処理用キューの仕様も含む.

(3) Label, Button などの組み込み UI 部品の ASM 仕様.

(4) 設計対象オブジェクトの ASM 仕様.
この部分を設計者が記述する.

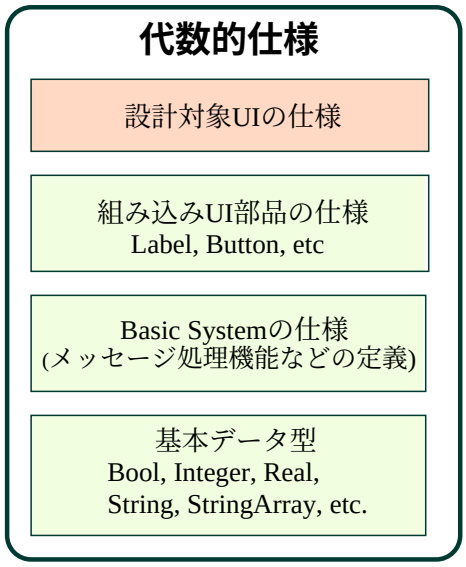


図 3.5　AWS の代数的仕様の構成

3.2 節で述べた例（SetValue オブジェクト）のカウンタ値を保持する機能などは，カウンタ値を表す状態成分関数を宣言し，各状態遷移の際のその成分の増減を公理で与えることで，ASM 仕様で容易に表すことができる．次に，メッセージ送受信に関する部分を考える．本仕様記述法では，メッセージ受信（メソッド呼出し）を，そのオブジェクトの状態遷移として表現する．そこで，あるオブジェクトがメッセージ受信の際に新たにメッセージを送信したい場合，あるオブジェクト自身の状態遷移の際に他のオブジェクトの状態遷移を起こす機能を定義できなければならない．しかし，各オブジェクトは AWS の一成分であり，それらの仕様を個別に記述するため，各オブジェクトは他のオブジェクトに行って欲しい状態遷移を指定する．それに対して AWS が,（メッセージ処理用キュー内のメッセージの処理を表す状態遷移を行ったとき）相手オブジェクトの状態を遷移させる．

行って欲しい状態遷移の指定は，特別な状態成分関数 “out” を使って記述するものとする．つまり，ある抽象的状態における out の値は，その状態に遷移したときに他のオブジェクトに対して要求する状態遷移である．このとき，行って欲しい状態遷移，つまり適用して欲しい状態遷移関数を out の関数値（送信先オブジェクトの指定なども必要なので，正確には関数値の一部）とする必要が生じるが，代数的仕様では関数を値として扱うことができない．関数を値として扱えるよう代数的仕様を拡張する方法も考えられるが，ここではより単純に，関数名を表す定数を各状態遷移関数ごとに用意して，関

数名定数を使って指定することにする．

メッセージの表現法

以降，他のオブジェクトに行って欲しい状態遷移を指定したものをメッセージと呼ぶ．メッセージは，送信先オブジェクトの名前，状態遷移関数名，および任意個の引数から構成される．例えば，親オブジェクトから n_val という名前の Label オブジェクトに setText という状態遷移を要求する場合，このメッセージは

msg(n_val , Label_setText , “hello”)

という項で表される．このメッセージは，オブジェクト n_val の抽象的状態を，状態 s から状態 setText(s , “hello”) に遷移させるよう要求している．ここで，n_val はソートが name である定数，Label_setText はソートが meth である定数である．関数を値として扱えるよう代数的仕様を拡張する方法（高階化）も考えられるが，ここではより単純に，関数名を表す定数を各状態遷移関数ごとに用意することにする（3.3.3 節参照）．オブジェクトの名前や関数名を表す定数は，公理で用いた分だけ自動的に宣言されると考える．また，関数 msg の引数の数や第 3 引数以降のソートはメッセージごとに変化するが，これも，適切な引数ソートを持つ msg が自動的に宣言されると考える（図 3.8 参照）．

AWS がこのメッセージを処理する（送信先オブジェクトの状態を遷移させる）際には，送信先オブジェクトを識別するため，送信先を名前ではなくオブジェクト ID で表す必要がある．上記メッセージの送信者である親オブジェクトの ID を j としたとき，送信先オブジェクトの ID は

child(j , n_val)

という項で表される．BasicSystem は，メッセージ処理用キューにメッセージを追加するとき，第 1 引数をオブジェクト ID に変換してから追加する．第 1 引数がオブジェクト ID であるメッセージは，

xmsg(child(j , n_val) , Label_setText , “hello”)

のように関数 xmsg を使って表す．

BasicSystem は，補助関数 apply を使って，このメッセージを状態遷移関数に対応づける．上の例の場合，この対応づけは以下のような公理で行われる．この公理は，3.3.2

節の形式 3 に当てはまらないが，xmsg の第 2，第 3 引数を取り出すような関数と if-then 関数を使えば，形式 3 の形で記述することができる．

forall o : oid, $param$: str, s : Label
ofsort Label
apply(xmsg(o , Label_setText , $param$) , s) = setText(s , $param$);

この公理は，状態 s である Label オブジェクトにメッセージ xmsg(o , Label_setText , $param$) を適用すると，この Label オブジェクトの状態が setText(s , $param$) に遷移するということを表している．この公理は状態遷移関数 setText の宣言（つまり引数と関数値のソート）のみから作ることができる．これらの公理の宣言も自動的に行われると考える．

BasicSystem の仕様

AWS は，各オブジェクトとメッセージ処理用キュー（キューと略記）を状態成分として持つ ASM である．すなわち AWS の抽象的状態は，全オブジェクトの状態およびキューの状態の組である．AWS の仕様中，メッセージ処理機能などの部分を BasicSystem と呼んでいる．BasicSystem の代数的仕様を図 3.6 に示す．また，キューの代数的仕様を図 3.7 に，メッセージを表すデータ型の代数的仕様を図 3.8 に示す．この他に，前述した，自動的に宣言されると考える関数（関数名を表す定数など）や公理が加わる．

BasicSystem の状態成分関数および状態遷移関数を以下に説明する．

- comp: オブジェクト ID が j であるオブジェクトの現在の状態を表す状態成分関数.
- mq: キューの現在の状態を表す状態成分関数.
- next: キュー内のメッセージを，キューが空になるまで処理する．下記の call などを使って，形式 2 の公理（3.3.2 節）で意味定義される（公理 (*B1)）.
- call: 1 つのメッセージを処理する．すなわち，メッセージで指定された状態遷移関数を m，引数として与えられたデータを a とすると，送信先オブジェクト o の状態を $m(o, a)$ に遷移させる（公理 (*B2)）．同時に，out$(m(o, a))$ をメッセージ処理用キューに追加する（公理 (*B3)）．out は，各オブジェクトが持つ，状態遷移時に送信するメッセージ列を表す状態成分関数である．関数 enqueue の第 3 引数は，第 2 引数であるメッセージ列の送信者の ID を表す．ここでは，メッセージ m の受信

```
type basicsystem is MessageQueue
  sorts state
  opns next, shift  : state           → state
       call, enq    : state, message  → state
       init_state   :                 → state
       comp         : state, oid      → obj
       mq           : state           → mqueue
  eqns
    forall s : state, m : message, j : oid
    ofsort state
      next(s) = if isempty(mq(s)) then s
                else next( call( shift(s) , head( mq(s) ) ) );                    (*B1)
    ofsort obj
      comp( call( s, m) , j ) = if receiver(m) eq j
                                then apply( m , comp( s , j ) )
                                else comp( s , j );                               (*B2)
      comp( enq( s , m) , j ) = comp( s , j );
      comp( shift(s) , j )    = comp( s , j );
    ofsort mqueue
      mq( call( s , m ) ) = enqueue( mq(s) ,
                                     out( apply( m , comp( s , receiver(m) ) ) ),
                                     receiver(m) );                               (*B3)
      mq( enq( s , m ) )  = enqueue( mq(s) , m );                                 (*B4)
      mq( shift(s) )      = dequeue( mq(s) );                                     (*B5)
      mq( init_state )    = empty_queue;                                          (*B6)
endtype
```

図 3.6 BasicSystem の代数的仕様

者が新しいメッセージ列の送信者である．送信者の ID は，メッセージの送信先をオブジェクト ID に変換するために用いられる．

- enq: AWS 外からのメッセージをキューの末尾に追加する（公理 (*B4)）．
- shift: キューの先頭要素を削除する（公理 (*B5)）．
- init_state: 初期状態を表す定数（公理 (*B6)）．

図 3.6 でのソート obj は，任意のオブジェクトの状態を表すソートである．各オブジェクトの状態を表すソートを obj に変換する関数が自動的に定義され，前述の関数 apply に対する公理は，実際には，Label などの個々のソートの値ではなく，このソート変換関数を適用した後の値を返すよう定義される．もし順序ソートを導入するなどして代数的仕様を拡張すれば，このようなソート変換関数は不要となる．

```
type MessageQueue is Message, MessageList, BOOLEAN
  sorts mqueue
  opns
      empty_queue  :                          → mqueue
      enqueue      : mqueue, message          → mqueue
      enqueue      : mqueue, message, oid     → mqueue (* with sender's id *)
      enqueue      : mqueue, mlist, oid       → mqueue (* with sender's id *)
      dequeue      : mqueue                   → mqueue
      head         : mqueue                   → message
      isempty      : mqueue                   → bool
  eqns
    forall q : mqueue, l : mlist, j : oid, m : message
    ofsort mqueue
      dequeue( enqueue( q , m ) ) = if isempty(q) then empty_queue
                                    else enqueue( dequeue(q) , m ) ;
      enqueue( q , l , j )  = if isnull(l) then q
                              else enqueue( enqueue( q , car(l) , j ) , cdr(l) , j ) ;
      enqueue( q , m , j )  = enqueue( q , resolv( m , j ) ) ;
    ofsort message
      head( enqueue( q , m ) ) = if isempty(q) then m else head(q) ;
    ofsort bool
      isempty( empty_queue ) = true ;
      isempty( enqueue( q , m ) ) = false ;
endtype
```

図 3.7 型 MessageQueue の代数的仕様

型 MessageQueue（図 3.7）では以下の関数が定義される．

- empty_queue: キューの初期状態を表す定数．
- enqueue: キューにメッセージを追加する関数．引数が 3 個の場合，第 3 引数はメッセージ送信を行なうオブジェクトの ID を表す．
- dequeue: 先頭メッセージを削除する関数．
- head: キューの先頭メッセージを得る関数．
- isempty: キューが空かどうかを得る関数．

型 Message（図 3.8）では，前述の（各引数ソートに対応する）msg，xmsg が宣言されている．msg や xmsg の第 3 引数以降は，第 2 引数であるメソッドごとに異なる．ここでは引数のないメソッドと，整数型の引数 1 個をとるメソッドに対応する例を示している．また，関数 receiver，resolv が定義されている．receiver は，メッセージの宛先で

```
type Message is
  sorts message
  opns
      xmsg      : oid, meth          → message
      msg       : name, meth         → message
      xmsg      : oid, meth, int     → message
      msg       : name, meth, int    → message
      ...
      receiver  : message            → oid
      resolv    : message, oid       → message (* translate a name into an oid *)
  eqns
    forall j : oid, g : message, v : int, n : name
    ofsort oid
      receiver( xmsg( j , g ) ) = j
      receiver( xmsg( j , g , v ) ) = j
      ...
      resolv( msg( n , g ) , j ) =
          if n eq parent then xmsg( parent(j ) , g)
          else if n eq self then xmsg( j , g )
          else xmsg( child( j , n ) , g )
      resolv( msg( n , g , v ) , j ) =
          if n eq parent then xmsg( parent(j ) , g , v )
          else if n eq self then xmsg( j , g , v  )
          else xmsg( child( j , n ) , g , v )
      ...
endtype
```

図 3.8 型 Message の代数的仕様

あるオブジェクト ID を表す関数である．resolv は，msg を使って表されたメッセージを xmsg を使ったものに変換する．すなわち，宛先を名前からオブジェクト ID に変換する関数である．resolv の第 2 引数は送信者のオブジェクト ID を表す．

組み込み UI 部品の仕様

組み込み UI 部品の仕様例として, Button オブジェクトと Label オブジェクトの ASM 仕様を図 3.9 に示す. ASM 仕様上では, 組み込み UI 部品や設計対象オブジェクトなどの区別はないので, 図 3.9 を設計対象オブジェクトの仕様記述例と考えてもよい.

AWS 上のオブジェクトの代数的仕様は, 以下の条件を満たす ASM 仕様である. これらの制約は, 3.4 節で述べるプロトタイプ生成において, ASM 仕様とプロトタイプ間で名前の対応などを単純にするためのものであり, ASM 仕様の形式的意味を変更するものではない.

- ASM 仕様の名前（キーワード type の右に書かれた Button や Label）と同じ名前の状態遷移関数は, オブジェクトの初期状態を表す （公理 (*T1)）.
- 前述のように, out は状態遷移時に送信すべきメッセージ列を表す状態成分関数である. メッセージ列を $[m_1, m_2, \ldots]$ と記述する（公理 (*T2)）.
- destroy は画面上の表示を消す状態遷移関数である. ASM 仕様では, out 以外の状態成分関数がすべて未定義の状態に遷移するように定義する（公理 (*T3)）.
- ASM 仕様 A 中で状態遷移関数 m を宣言した場合, m の関数名を表すソート meth の定数 A_m も宣言されているとみなす. （定数 A_m はメッセージを構成する関数 msg や xmsg の第 2 引数に用いる.）

仕様では, ユーザからの直接操作と他オブジェクトからのメッセージ受信に対して，それぞれ状態遷移関数を用意し公理を記述する. Button オブジェクトの仕様では，press 関数が前者，Button，destroy 関数が後者にあたる.

```
type Button is Widget
  sorts button
  opns Button        : str, meth → button                    (*T1)
       press, destroy : button    → button
       out            : button    → mlist (* output message *)
       callback       : button    → meth
  eqns
    forall b : button, g : meth, x : str
    ofsort mlist
      out( Button( x , g ) ) = [ ];                          (*T2)
      out( press(b) )        = [ msg( parent , callback(b) ) ];
      out( destroy(b) )      = [ ];                          (*T3)
    ofsort meth
      callback( Button( x , g ) ) = g
      callback( press(b) )        = callback(b)
endtype

type Label is Widget
  sorts label
  opns
    Label   : str    → label                                 (*T1)
    setText : label  → label
    destroy : label  → label
    out     : label  → mlist  (* output message *)
    disp    : label  → str    (* display string *)
  eqns
    forall l : label, x : str
    ofsort mlist
     out( Label(x) )        = [ ];                          (*T2)
     out( setText( l , x ) ) = [ ];
     out( destroy(l) )      = [ ];                          (*T3)
    ofsort str
     callback( Label(x) )        = x
     callback( setText( l , x ) ) = x
endtype
```

図 3.9 ASM 仕様記述例：Button, Label

3.3.4 UIの仕様記述例

3.2 節で述べた SetValue の ASM 仕様を図 3.10 に示す．抽象的状態を表すソート名は setvalue である．

状態遷移関数として, 初期状態を表す SetValue, 4 つのボタンのクリックをそれぞれ表す up, down, reset, ok および自分を消滅させる destroy が宣言されている．SetValue(x , g , v) は，「ラベルとして x が画面上に表示され, コールバックメソッドとしてメソッド g が指定され, カウンタ値が v である」初期状態を表す．状態成分関数として, コールバックメソッドを表す callback, 現在のカウンタの値を表す get, カウンタの初期値を表す getinit, 送信すべきメッセージ列を表す out が宣言されている．次に公理について説明する．公理は状態成分関数ごとにまとめられている．もちろん，状態遷移関数ごとにまとめるなど，他の並び方に変えても仕様の意味は変化しない．

(*S1)–(*S4) は状態成分関数 out に関する公理である．公理 (*S1) は，この UI は初期状態 SetValue(s , g , v) において, 4 つの Button オブジェクトおよびカウンタ値を表示するための 2 つの Label オブジェクトの合計 6 つの部品を生成するためのメッセージを送信することを表す．例えば msg(n_up , Button_Button , “up” , SetValue_up) は, タイプ Button の新しいオブジェクト n_up 宛に，その初期状態を表す状態遷移関数に対応するメソッド名 Button のメッセージを送信することを表す．“up” と SetValue_up はそのメソッドの引数であり, “up” は生成されるオブジェクト n_up に付けられる画面上のラベルを表し, SetValue_up は n_up からのコールバックメソッド名を表す．同様に，公理 (*S2) では，状態遷移 up が起こると, 遷移前のカウンタ値 get(s) に 1 加えた値を引数として, Label オブジェクト n_val に対してメソッド setText を送信すること，公理 (*S3) では，状態遷移 ok が起こると, 遷移前の値 get(s) を引数として親オブジェクト **parent** に対してメソッド名 callback(s) のメッセージを送信すること，公理 (*S4) では，状態遷移 destroy が起こると各 UI 部品のオブジェクトに対してそれらを消滅させるためのメッセージを送信することを記述している．公理 (*S5)–(*S8) は，状態成分関数 get に関する公理である．これらの公理により get の値は，初期状態 SetValue(s , g , v) においてはその入力パラメタ v に初期化され，状態遷移 up が起こると遷移前の値に 1 を加えた値 (公理 (*S6))，状態遷移 reset が起こると初期値 getinit(s)(公理 (*S7)) に変更されることが記述されている．一方，状態遷移 ok が起こっても状態遷移前と値が変わらないこと (公理 (*S8)) が記述されている．

SetValue を子オブジェクトとして利用する Example の ASM 仕様を図 3.11 に示す．状

態遷移関数として，初期状態を表す Example，子オブジェクトから値を受信し処理をする done， このオブジェクト自身を消滅させる destroy が宣言されている．状態成分関数として，SetValue と同様に送信するべきメッセージ列を表す out が宣言されている．公理では，初期状態 Example において n_setv という名前を持つ SetValue 型の子オブジェクトを生成すること，状態遷移 done を行なったときは，自分自身 **self** に対して状態遷移 destroy を引き起こすメッセージを送信することなどが記述されている．

3.4 コンパイラの実現と変換例

3.4.1 プロトタイプ生成の方針

本仕様記述法で導入した AWS モデルは，一般的なオブジェクト指向ツールキットに基づくものとなっており，現在利用可能なツールキットの多くは，プロトタイプを実装するためのアーキテクチャとして使用できると考えられる．今回は可搬性の高さなどから，Java[34] および Swing ツールキット [31] を実装アーキテクチャとして選んだ．図 3.12 に，3.3 節で述べた代数的仕様とプロトタイプとの対応関係を示す．今回作成したプロトタイプ生成用コンパイラは，設計者が記述した設計対象 UI の仕様を入力とし，その実装となっているような Java プログラムを出力するものである．具体的には，一つの ASM 仕様が Java における一つのクラスに，各状態遷移関数がそのクラスのメソッドに，各状態成分関数がそのクラスのデータメンバに，それぞれ変換される．状態遷移関数・状態成分関数の変換方法は [9] と同様である．ただし，状態遷移時に送信するメッセージ列を表す状態成分関数 out は，データメンバに変換せず，後述のような特別な扱いをする．

設計対象 UI 以外の部分は以下のように扱う．

基本データ型は，Java の int 型や String クラスなどのデータ型・クラスに対応づける．つまり，設計対象 UI の仕様中に現れる基本データ型の演算に対して，Java のデータ型・クラスで定義された演算がその実装になっていると考え，これらの演算への変換を行う．

BasicSystem の機能のうち，オブジェクト間のメッセージ送受信機能は，Java での通常のメソッド呼出しによって実現する．例えば「SetValue オブジェクトが状態遷移 up を行った際，名前が n_val である子オブジェクトに setText メッセージを送信する」という仕様は，Java による実装においては,「SetValue クラスのメソッド up の中で，データメンバ n_val が指すオブジェクトのメソッド setText を呼び出す」という形で実現される．具体的には，状態遷移関数 up の実装であるメソッド内で out 以外の状態成分に対する更

```
type SetValue is frame
    sorts setvalue
    opns
        SetValue              : str, meth, int → setvalue
        up, down, reset, ok   : setvalue       → setvalue
        destroy               : setvalue       → setvalue
        callback              : setvalue       → meth
        get, getinit          : setvalue       → int
        out                   : setvalue       → mlist
    eqns
        forall s : setvalue, x : str, g : meth, v : int
        ofsort mlist
            out( SetValue( x, g, v ) ) =
                [ msg( n_tag ,   Label_Label,     x ++ ":" ),
                  msg( n_val ,   Label_Label,     String(v) ),
                  msg( n_up ,    Button_Button,   "up" ,     SetValue_up ),
                  msg( n_down ,  Button_Button,   "down" ,   SetValue_down ),
                  msg( n_reset , Button_Button,   "reset" ,  SetValue_reset ),
                  msg( n_ok ,    Button_Button,   "ok" ,     SetValue_ok )
                ];                                                              (*S1)
            out( up(s) )       = [ msg( n_val , Label_setText , String( get( up(s) ) ) ) ];   (*S2)
            out( down(s) )     = [ msg( n_val , Label_setText , String( get( down(s) ) ) ) ];
            out( reset(s) )    = [ msg( n_val , Label_setText , String( get( reset(s) ) ) ) ];
            out( ok(s) )       = [ msg( parent , callback(s) , get(s) ) ];      (*S3)
            out( destroy(s) )  = [ msg( n_tag ,   Label_destroy ),
                                   msg( n_val ,   Label_destroy ),
                                   msg( n_up ,    Button_destroy ),
                                   msg( n_down ,  Button_destroy ),
                                   msg( n_reset , Button_destroy ),
                                   msg( n_ok ,    Button_destroy )
                                   ];                                           (*S4)
```

```
ofsort int
    get( SetValue(x, g, v))  = v ;                          (*S5)
    get( up(s))              = get(s) + 1 ;                 (*S6)
    get( down(s))            = get(s) - 1 ;
    get( reset(s))           = getinit(s) ;                 (*S7)
    get( ok(s))              = get(s) ;                     (*S8)
    getinit( SetValue( x , g , v ) ) = v ;
    getinit( up(s) )                 = getinit(s) ;
    getinit( down(s) )               = getinit(s) ;
    getinit( reset(s) )              = getinit(s) ;
    getinit( ok(s) )                 = getinit(s) ;
ofsort meth
    callback( SetValue( x , g , v ) )  = g ;
    callback( up(s) )                  = callback(s) ;
    callback( down(s) )                = callback(s) ;
    callback( reset(s) )               = callback(s) ;
    callback( ok(s) )                  = callback(s) ;
endtype
```

図 3.10 ASM 仕様記述例：SetValue

```
type Example is
    sorts
        example
    opns
        Example :                 → example
        done    : example, int → example
        destroy : example      → example
        out     : example      → mlist
    eqns
        forall s : example, v : int
        ofsort mlist
            out(Example) =
                [ msg( n_setv , SetValue_SetValue , "example", Example_done , 0 ) ];
            out( destroy(s) )       = [ msg( n_setv , SetValue_destroy ) ];
            out( done( s , v ) )    = [ msg( self , Example_destroy ) ];
endtype
```

図 3.11 ASM 仕様記述例 : Example

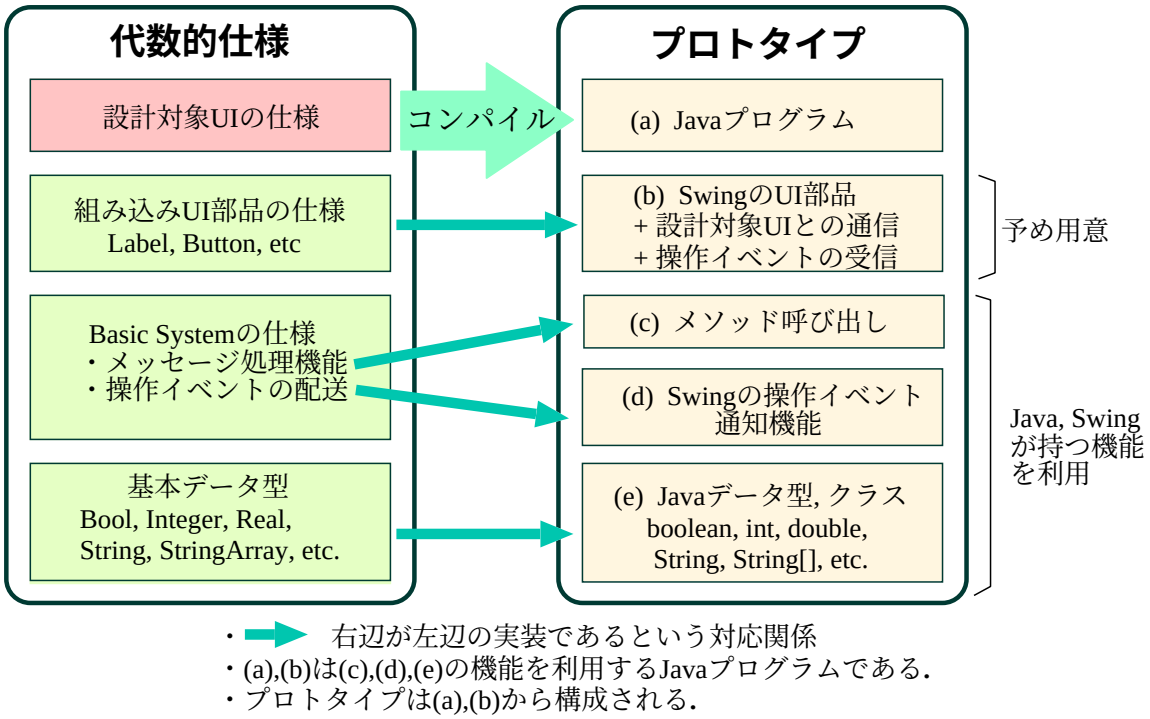


図 3.12 代数的仕様とプロトタイプとの対応関係

新を行った後，out で指定されたメッセージ列の送信を表すメソッド呼出し列を実行する．子オブジェクトの生成も，親オブジェクトのメソッド内で Java での通常のオブジェクト生成を行うことで実現する．作成された子オブジェクトへの参照を，親オブジェクトがデータメンバとして保持する．一方，親へのメッセージ送信を行うために，各オブジェクトは自分の親オブジェクトを知っておく必要がある．そこで実装上では，各生成メソッドは仕様で宣言された引数に加えて，親オブジェクトへの参照を引数として受け取ることにし，親が子の生成メソッドを呼び出す際に自分への参照を渡すようにする．

AWS 外からのメッセージの受信，すなわち操作イベントの受信は，各組み込み UI 部品の実装において個別に実現する．すなわち，Swing が提供する操作イベント通知機能を使って，仕様に書かれた動作を実行するメソッドが操作イベント発生時に呼び出されるようにしておく[1]．

各組み込み UI 部品について，Java における一つのクラスの形でその部品の実装を与

[1]今回作成したプロトタイプ生成用コンパイラへの入力では，Swing が提供する操作イベント通知機能と ASM 仕様上の状態遷移関数との対応を記述できないため，操作イベントを直接受信するオブジェクトは組み込み UI 部品に限られる．

えておく．設計対象オブジェクトとのメッセージ送受信を行えるよう，設計対象オブジェクトと同様に，各状態遷移関数をメソッドとして，メッセージの送信を送信先オブジェクトに対するメソッド呼出しとしてそれぞれ実装する．通常は，Swing で提供されている UI 部品クラスを利用し，メソッドの引数の型を仕様上の状態遷移関数の引数のソートに合わせるためのコードや，上述の，操作イベント通知に対して対応するメソッドを呼び出すためのコードなどを加える．

3.4.2 コンパイラの概要

3.4.1 節の方針に基づくコンパイラを Java プログラムとして実装した．字句解析部および構文解析部の作成には，字句解析器生成ツール JLex および構文解析器生成ツール Cup[35] をそれぞれ用いた．コンパイラプログラムのサイズは，JLex, Cup への入力である構文規則および意味動作の記述が約 700 行，コンパイラ内部で扱うデータ構造（クラス）の定義が約 1,400 行，コード生成部などが約 650 行である．

3.4.3 適用例 1

3.3.4 節で示した仕様記述例（図 3.10）をコンパイルし得られた Java プログラムから特長的な部分を抜粋し紹介する．コンパイル結果の一部を図 3.13 に示す．

図 3.13(a) は，状態遷移関数 ok に対応するメソッドのプログラムコードである．ASM 仕様上での callback という状態成分関数は，実装上では `Method` クラス[2] のデータメンバ `callback` で表される．

親オブジェクトに対するメッセージ msg(**parent** , callback(s) , get(s)) の送信は，親オブジェクトを記憶しておくための `Object` クラスのデータメンバ `parent` を用いて，

```
Object[] args = { new Integer(prev_get)};
prev_callback.invoke(parent,args);
```

と表される．ただし，`prev_get`, `prev_callback` は状態遷移前の get, callback の値を保持する変数で，このメソッドの冒頭でそれぞれ get，callback の値が代入されている．図 3.13(b) は，初期状態を表す状態遷移関数 SetValue に対応するメソッドのプログラムコードの一部である．メソッド内の初めの 4 行は，SetValue を左辺に含む形式 1 の各公理 (3.3.2 節) に基づいて状態成分を表す各データメンバに初期値を代入している．5 行目

[2] java.lang.reflect.Method クラス.

```
public void ok() {
    int         prev_get         = this.get;
    int         prev_getinit     = this.getinit;
    Method      prev_callback    = this.callback;
    this.get      = prev_get;
    this.getinit  = prev_getinit;
    this.callback = prev_callback;
    try  {
         Object[] args = { new Integer(prev_get) };
         prev_callback.invoke(parent,args);
    } catch (InvocationTargetException e) {
    throw new RuntimeException(e.getMessage());
    } catch (IllegalAccessException e) {
    throw new RuntimeException(e.getMessage());
    }
}
```

(a) 生成されたメソッド ok

```
public SetValue(Object parent,
    String arg0,Method arg1,int arg2) {
        this.parent   = parent;
        this.get      = arg2;
        this.getinit  = arg2;
        this.callback = arg1;
        Container cp  = this.getContentPane();
        cp.setLayout(new FlowLayout());
        n_tag  = new Label(this,arg0+``:''  );
        cp.add(n_tag);
        n_val  = new Label(this,String.valueOf(arg2));
        cp.add(n_val);
        n_up   = new Button(this,``up'',up);
        cp.add(n_up);
        n_down = new Button(this,``down'',down);
        cp.add(n_down);
        n_reset= new Button(this,``reset'',reset);
        cp.add(n_reset);
        n_ok   = new Button(this,``ok'',ok);
        cp.add(n_ok);
        ...
```

(b) 生成されたメソッド SetValue の一部

図 3.13 コンパイル結果：SetValue.java（一部）

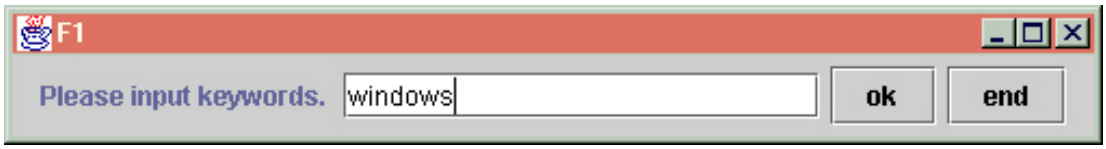


図 3.14　F1 の表示例

以降は，out の仕様に基づいて子オブジェクトの生成を行なっている．同時に，生成されたオブジェクトをウィンドウ上の構成分子として登録する作業も行っている．

3.4.4　適用例 2

1 タスクを 1 つのウィンドウで実現する次のような UI に対して本手法を適用した例を紹介する．

この例は以下の機能を持つ．

- ウィンドウ F1（図 3.14）
  - 入力促進文，入力欄と 2 つのボタン ok, end を表示する．
  - ボタン ok をクリックすると，キーワードが入力されている場合，ウィンドウ F1 を消去しウィンドウ F2 を表示する．このときキーワードを F2 に渡す．
  - ボタン end をクリックすると，ウィンドウ F1 を消去しシステムを終了する．
- ウィンドウ F2（図 3.15）
  - 説明文と 2 つのボタン ok, quit を表示する．
  - キーワードを元に検索した結果の本のリストをチェックボックスと共に表示する．
  - ボタン ok をクリックすると，チェックボックスが 1 以上チェックされている場合，ウィンドウ F2 を消去し，チェックされた本のデータを F3 に渡す．
  - ボタン quit をクリックすると，ウィンドウ F2 を消去しシステムを終了する．

図 3.16 はこの例のオブジェクトの構成である．F1 オブジェクトは 2 個の Button オブジェクトと入力促進文である Label オブジェクト，入力欄である TextField オブジェクト，そして，次のウィンドウのオブジェクトである F2 オブジェクトを構成分子として含

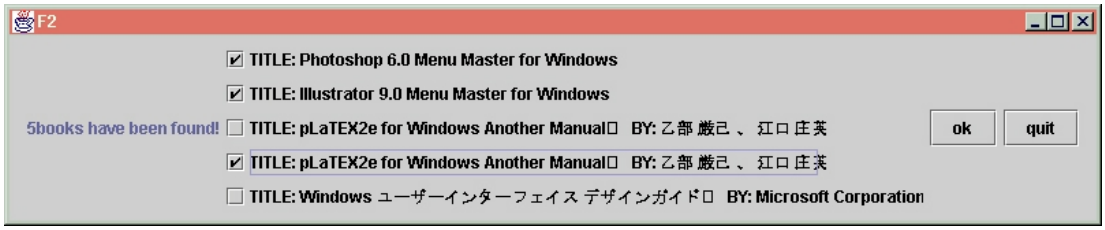


図 3.15　F2 の表示例

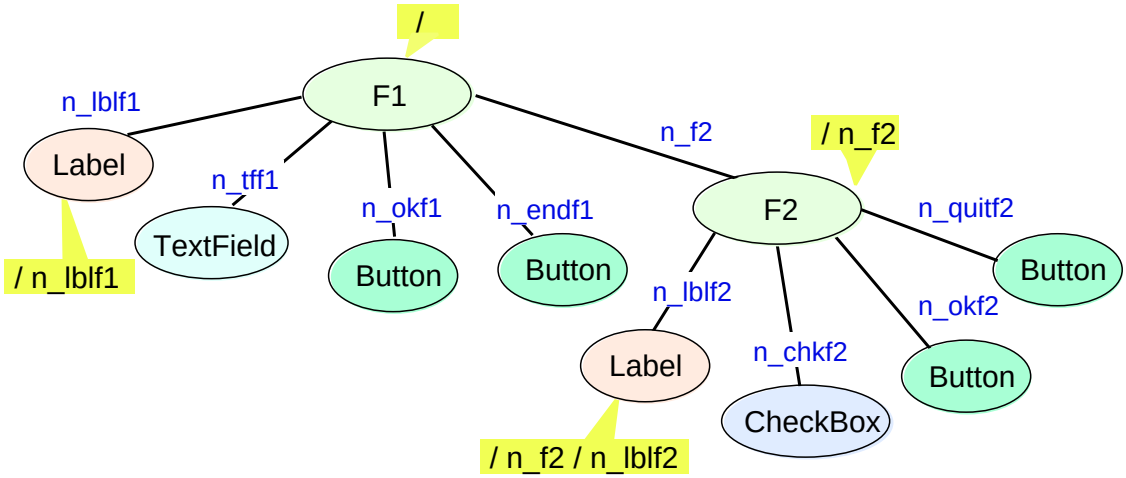


図 3.16　F1, F2 のオブジェクト構成

む．また，F2 オブジェクトは 2 個の Button オブジェクトと結果出力文である Label オブジェクト，入力欄である CheckBox オブジェクトを構成分子として含む．

この例の ASM 仕様を図 3.17，図 3.18 に示す．また，この例で利用される組み込み UI 部品のオブジェクト，TextField オブジェクトと CheckBox オブジェクトの ASM 仕様を図 3.19，図 3.20 に示す．

```
type F1 is frame
    sorts f1
    opns
        F1                  :            → f1
        textfield           : f1, str    → f1
        ok, end, destroy    : f1         → f1
        out                 : f1         → mlist
    eqns
        forall s : f1, k : str
        ofsort mlist
            out(F1) =      [ msg( n_lblf1 , Label_Label,           "Please input keywords. " ),
                             msg( n_tff1 ,  TextField_TextField,  " ",     F1_textfield ),
                             msg( n_okf1 ,  Button_Button,        "ok",    F1_ok ),
                             msg( n_endf1 , Button_Button,        "end",   F1_end )
                             ];                                                          (*F1-1)
            out( ok(s) ) = [ msg( n_tff1,   TextField_getField ) ] ;                     (*F1-2)
            out( textfield( s , k ) ) = if k ⟨⟩ " "
                                        then [ msg( n_f2 ,  F2_F2 , k ),
                                               msg( self ,  F1_destroy ) ];              (*F1-3)
            out( end(s) ) =      [ msg( self ,     F1_destroy ) ] ;                      (*F1-4)
            out( destroy(s) ) = [ msg( n_lblf1 , Label_destroy ),                        (*F1-5)
                                  msg( n_tff1 ,  TextField_destroy ),
                                  msg( n_okf1 ,  Button_destroy ),
                                  msg( n_endf1 , Button_destroy )
                                  ] ;                                                    (*F1-5)
    endtype
```

図 3.17 ASM 仕様例：F1

```
type F2 is frame
    sorts f2
    opns
        F2                  : str           → f2
        chk                 : f2, strarray  → f2
        ok, quit, destroy   : f2            → f2
        out                 : f2            → mlist
    eqns
        forall s: f2, k: str, b: strarray
        ofsort mlist
            out( F2(k) ) =
                [ msg( n_lbl2,    Label_Label,
                                  if length( search(k) ) ⟨⟩ 0
                                  then String( length( search(k) ) ) ++ "books have been found!"
                                  else "Books have not been found." ),
                  msg( n_chkf2 , CheckBox_CheckBox , search(k) , F2_chk ),
                  msg( n_okf2 ,  Button_Button , "ok" , F2_ok ),
                  msg( n_quitf2 ,Button_Button , "quit" , F2_quit )
                  ];                                                          (*F2-1)
            out( ok(s) ) =      [ msg( n_chkf2 , CheckBox_getCheckList ) ];   (*F2-2)
            out( chk( s , b ) ) = if length(b) ⟨⟩ 0 then
                                    [ msg( n_f3 , F3_F3, b ),
                                      msg( self , F2_destroy )
                                      ];                                      (*F2-3)
            out( quit(s) ) =      [ msg( self ,      F2_destroy ) ];
            out( destroy(s) ) =   [ msg( n_lblf2 ,   Label_destroy ),
                                    msg( n_chkf2 ,   CheckBox_destroy ),
                                    msg( n_okf2 ,    Button_destroy ),
                                    msg( n_quitf2 ,  Button_destroy )
                                    ];
        endtype
```

図 3.18　ASM 仕様例：F2

```
type TextField is Widget
    sorts TextField
    opns
        TextField     : str, meth  → textfield
        getField      : textfield  → textfield
        destroy       : textfield  → textfield
        out           : textfield  → mlist
        callback      : textfield  → meth
        paramString   : textfield  → str
    eqns
        forall l : textfield, g : meth, x : str
        ofsort mlist
            out( TextField( x , g ) )= ⟨⟩ ;
            out( getField( l ) ) = [ msg( parent , callback(l) , paramString(l) ) ] ;      (*TF-1)
            out( destroy(l) ) = ⟨⟩ ;
        ofsort meth
            callback( TextField( x , g ) ) = g ;
            callback( getField(l) ) = callback(l) ;
    endtype
```

図 3.19　ASM 仕様例：TextField

F1 の ASM 仕様（図 3.17）では，状態遷移関数として，初期状態を表す F1，TextField オブジェクトからの入力テキストの受信を表す textfield，2 つのボタンのクリックを表す ok, end, および自分を消滅させる destroy が宣言されている．状態成分関数としては，送信すべきメッセージ列を表す out のみが宣言されている．公理 (*F1-1) では，この UI は初期状態 F1 において，入力促進文を表示するための Label オブジェクト，キーワードを入力するための TextField オブジェクト，そして，2 つの Button オブジェクトの合計 4 つの部品を生成するためのメッセージを送信することを表す．公理 (*F1-1) 右辺の 2 行目より，TextField オブジェクトへのコールバックメソッド名として textfield を指定していることに注意されたい ($\star$)．公理 (*F1-2) は，ok ボタンをクリックしたとき，TextField オブジェクト n_tff1 に対して getField メッセージを送信することを表す．TextField オブジェクト（図 3.19）において，状態遷移 getField が起こると，親オブジェクト **parent** すなわち F1 オブジェクトに対して，paramString(l) を引数とした callback(l) メッセージを送信する（公理 (*TF-1)）．($\star$) で述べたことより，callback(l) は実際には F1 オブジェクトの状態遷移関数 textfield であるので，callback(l) メッセージの F1 オブジェクトへの配送により状態遷移関数 textfield が適用される．公理 (*F1-3) より，F1 オブジェクトは，テキスト入力欄の値 k が空でない場合，k を引数として F2 オブジェクトを生成し，

```
type CheckBox is Widget
    sorts checkbox
    opns
        CheckBox      : strarray, meth→ checkbox
        getCheckList  : checkbox, str  → checkbox
        destroy       : checkbox       → checkbox
        out           : checkbox       → mlist
        callback      : checkbox       → meth
        checklist     : checkbox       → strarray
    eqns
        forall l : checkbox, x : str, g : meth
        ofsort mlist
            out( CheckBox( x , g ) ) = ⟨⟩ ;
            out( destroy(l) ) = ⟨⟩ ;
            out( getCheckList(l) ) = [ msg( parent , callback(l) , checklist(l) ) ] ;   (*CB-1)
        ofsort meth
            callback( CheckBox( x , g ) ) = g ;
            callback( getCheckList(l) ) = callback(l) ;
    endtype
```

図 3.20　ASM 仕様例：CheckBox

F1 オブジェクト自身を消滅することによりウィンドウの遷移を実現する．

F2 の ASM 仕様（図 3.18）では，状態遷移関数として，初期状態を表す F2，CheckBox オブジェクトからのデータの受信を表す chk，2 つのボタンのクリックを表す ok, quit および自分を消滅させる destroy が宣言されている．状態成分関数として，送信すべきメッセージ列を表す out が宣言されている．公理 (*F2-1) は，この UI は初期状態 F2 において，入力促進文を表示するための Label オブジェクト，チェックボックスにて本を 0 冊以上選択するための CheckBox オブジェクト，そして，2 つの Button オブジェクトの合計 4 つの部品を生成するためのメッセージを送信することを表す．さて，公理 (*F2-1) の右辺に現れている search は変数 k をキーワードとして本を検索する補助関数であり，検索された本の配列を関数値として返す．F2 オブジェクトは，その配列の長さが 0 以外のとき，すなわち，該当する本の冊数が 0 以外のとき冊数を表示し，0 のとき見つからなかった旨を表示する．公理 (*F2-2) は，ok ボタンをクリックしたとき，CheckBox オブジェクトに getCheckList メッセージを送信することを表す．CheckBox オブジェクト（図 3.20）において，状態遷移関数 getCheckList が適用されると，親オブジェクト **parent** すなわち F2 オブジェクトに対して，checklist(l) を引数とした callback(l) メッセージを送信する（公理 (*CB-1)）．公理 (*F2-1) 右辺の 5 行目より，callback(l) は実際には F2 オブジェ

クトの状態遷移関数 chk であるので，callback(l) メッセージの F2 オブジェクトへの配送により，状態遷移関数 chk が適用される．公理 (*F2-3) より，F2 オブジェクトは，チェックされているチェックボックスのリストである b が空でない場合，F3 という別のウィンドウを生成して b を渡し（F3 についての説明は省略），F2 オブジェクト自身を消滅する（公理 (*F2-3)）．

F1 と F2 の ASM 仕様において状態成分関数はそれぞれ 1 個と 2 個，また状態遷移関数は共に 5 個である．ただし，F1 や F2 が生成する Button オブジェクトなどの ASM 仕様は含まない．コンパイルおよびプログラム実行には DEC PersonalWorkstation 500au (Alpha21164A(500MHz), 128MB RAM) を使用した．この例での ASM 仕様のコンパイル時間は 1.97 秒であり，生成された Java プログラムの実行において待ち時間は殆んどなかった．

3.5 第 3 章のまとめ

本研究では，AWS モデルに基づいた UI の ASM 仕様の記述法を提案した．この記述法により，UI 部品間の関係や画面遷移，データの扱いなど，UI の設計内容を明確に記述できることを示した．また，その ASM 仕様を Java プログラムに変換するコンパイラと，コンパイル例を示した．

今後の課題として，UI の ASM 仕様に対する形式的検証が挙げられる．また，今回考慮しなかった UI 部品（ラジオボタン，コンボボックスなど）や前のウィンドウに戻る・中断・割り込みなどの構造を記述できるような ASM 仕様について検討し，それに対応するようコンパイラを改善することが考えられる．

第4章　結論

4.1　まとめと考察

本研究では，インタラクティブシステムを対象とした設計の上流工程から適用できる設計法を提案した．ユーザの作業の流れや作業中に用いるデータをタスクモデルとして記述することにより，ユーザタスクの構造や流れを重視した設計が可能となると考えられる．図的記述法であるタスク図を仕様記述に用いることにより，直観的にユーザの作業手順を表現することができ，従来の形式的記述言語を仕様記述に直接用いた場合に比べて仕様の作成や理解が容易になると思われる．また，タスク図の意味を，プロセス代数に基づく記述法である LOTOS[5, 6] を用いて形式的に定義した．タスクモデルを段階的に詳細化することにより，詳細化前後の設計のプロセス間の一貫性を保つことができ，仕様の正当性が保証できる．次に，LOTOS 仕様に対するモデル検査法を用いて，タスク図で記述されたタスクモデルを形式的に検証する手順について述べた．これにより，タスクモデルの問題を発見し，その修正を行ない，そして再び形式的検証を繰り返し，タスクモデルを洗練化することが可能となった．さらに，タスクモデルからプロトタイプを自動生成する仕組みを提案した．システムのユーザビリティは，プロトタイプをテストすることにより評価される．プロトタイプ生成を自動化することにより，プロトタイプ作成の際の誤りを減少させることができる．図的記述法であるタスク図を用いるため，コード列で記述する場合に比べ，問題の箇所を特定し易いと思われる．したがって，プロトタイプのテスト結果をタスクモデルに迅速に反映でき，システムの品質向上のための作業の効率化が図られ，コスト削減につながると考えられる．

また，UI を代数的仕様記述の部分クラスである抽象的順序機械型仕様 (ASM 仕様) を用いて形式的に記述する方法を提案した．ASM 仕様を記述するにあたり，個々の UI モジュールがメッセージ送受信によって非同期に動作するような，抽象的ウィンドウシステム（AWS）モデルを導入した．基本データ型や AWS の各機能を実現する部分，Label や Button などの組み込み UI 部品の仕様を予め作成しているため，設計者は各 UI 部品を用

いてメッセージを送受信する部分についての記述を行なうだけで良い．そして，Java を実装アーキテクチャとした UI の ASM 仕様を実装するための枠組を提案し，これにしたがって ASM 仕様を Java プログラムに変換するコンパイラを作成した．異なる実装アーキテクチャに対しても，このコンパイラの枠組にしたがって同様の機能を持つプロトタイプを生成することが可能である．

4.2 今後の展望

従来，システム開発手法は、特定分野（製造業，金融業等）を対象にしたシステム開発と共に発展してきた．一方，インターネットが急速に広く社会へ浸透した現在では，パソコンやモバイル端末を用いて誰でも手軽に Web を利用できるようになってきている．例えば，単純なボタン操作などだけで目的が達成できていた券売機や家電などの機器類も，画面遷移をともなう多機能なシステムに変わってきている．Web アプリケーション技術は，従来の特定分野を対象にしたシステムにも影響を与えつつある．使用される場面やユーザが多様化することにより，ユーザ要求仕様は複雑化してきている．システム開発において，特にインタラクティブ設計部分は大規模・複雑化しており，システム製造工程はもちろん，設計工程の作業の正確さが要求されてきている．一般に，形式的手法の利用により信頼性の高いソフトウェアが提供可能となっている．しかし，高度な数学的知識が必要であり経済性に乏しいとされているため，特定分野の特定領域に適応されることが多い．一般のシステム製造に携わるエンジニアでも比較的容易に利用することができ，システム開発の上流工程で利用できる形式的手法の必要性がますます高くなってくると考える．

本研究では，ソフトウェアをビジネスロジック層，データ層，プレゼンテーション層の 3 層構造として捉えシステム化の対象領域に依存しない設計法を提案した．さらに，プレゼンテーション層にかかわる部分を交えた形式的手法をシステム開発の上流工程から適用することにより，UI を含んだシステム全体を対象とした形式的検証を可能とする設計法を示した．それにより，UI 部品の利用とユーザタスクとの矛盾を未然に防ぐことが可能となる．また，プロトタイプの自動生成により効率的な開発が可能となり，システム実装時のプログラミングエラーを防ぐことができる．

今回提案したインタラクティブシステム設計法では，本設計法を説明するために「通信販売で本の購入」といった単純なモデルを採用した．この例題は，情報の入力・処理・

出力という基本的な流れを含んでおり，小規模ではあるが本設計法の実用性を示すことができた．

本設計法を評価・展開していくためには，大規模システム開発への適用や，既存システムの拡張等において既存タスクや UI の再利用を考慮したシステム開発への適用や評価が必要と考えられる．また，本設計手法の有効性を評価するためには，タスクモデルの再構成や UI 部品の組合せを行うことにより，開発されるシステムの有効性やコスト評価等を行うことが考えられる．また，既に提供されているアプリケーションで用いられているタスクや UI を再利用するなどのシステムの拡張に本手法を適用し，その効果について考察することが考えられる．一方，本手法を用いると，システムを実現する機能は同じだが，異なるタスク構造を持つタスクモデルを複数記述することができる．それにしたがって自動生成したプロトタイプのうち，ユーザが望むものを選択できる方法も提供できると考える．

今後，システムを利用するユーザのメタファや利用する環境などは思いもよらない状況で変化するであろう．急速に高度に進化しつつあるシステムに対して，ユーザの要求を正確かつ柔軟，迅速にシステムに実現する仕組みの探求は重要課題であり今後の発展が期待される．

謝辞

本研究を行う機会を与えて頂き，研究の全過程において直接懇切なる御指導御鞭撻を賜りました情報基礎学講座 関 浩之 教授に深く感謝の意を表します．システム開発の現場とは異なる次元から “情報” という世界を捉える方法を学ぶことができました．

本研究の全過程を通して有益な御助言と励ましの言葉をいただきました言語設計学講座 渡邉 勝正 教授，ソフトウェア計画構成学講座 松本 健一 教授に心より感謝の意を表します．

大学院生活全般に関して様々な御助言を与えて下さいました研究科長 植村 俊亮 教授に心より感謝の意を表します．

本研究の全過程を通して終始的確な御助言を与えて下さいました情報基礎学講座 楫 勇一 助教授，そして，直接御指導下さいました情報基礎学講座 高田 喜朗 助手に深く感謝の意を表します．また，研究生活を送る上で様々な御支援をいただきました関研究室の諸氏に深く感謝致します．

研究全般にわたり有益な御助言と，学費獲得のためのきっかけを与えて下さいました四国学院大学文学部英文学科 日尾 康子 教授，本学ソフトウェア計画構成学講座 島 和之 助手に厚く御礼申し上げます．また，非常勤講師としての授業遂行を応援していただきました四国学院大学文学部英文学科の教職員の皆様，大阪樟蔭女子大学情報処理研究室の皆様に深く感謝致します．

暖かい思いやりと励ましの心遣いを始終いただきました友人 岩井 幸子 女史に深く感謝致します．また，和香会（茶道裏千家）の皆様の日本古来の伝統文化の教えにより，情報科学とは対極の視点から思索しバランス感覚を保つことができました．厚く御礼申し上げます．

本研究を遂行するにあたり多くの方々に御支援をいただきました．皆様に深く感謝致します．

最後に，研究する機会と多方面からの助言，そして，激動する会社情勢の中にもかかわらず，生活基盤を強固に保ち，安心して学位取得に励むための環境を提供してくれた，本学先輩でもある夫 池田 吉隆 には感謝はもちろんのこと，多大なる尊敬の念を抱かざるを得ません．

参考文献

[1] Newman, W. M. and Lamming, M. G.: *Interactive System Design*, Addison-Wesley (1995).

[2] Microsoft: *Microsoft Visual Basic 6.0 Programmer's Guide*, Microsoft Corporation (1998).

[3] 服部誠: Borland Delphi 5 公式コースウェアシリーズ (基礎編) Inprise 公式コースウェアシリーズ, アスキー出版局 (1999).

[4] Booch, G., Rumbaugh, J. and Jacobson, I.: *The Unified Modeling Language User Guide*, Addison-Wesley (1999).

[5] ISO: Information Processing System, Open Systems Interconnection, LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO 8807 (1989).

[6] 高橋薫, 神長裕明: 仕様記述言語 LOTOS, カットシステム (1995).

[7] Gognen, J. A. and Malcolm, G.(eds.): *Software Engineering with OBJ*, Kluwer Academic Press (2000).

[8] 杉山裕二, 谷口健一, 嵩忠雄: 基底代数を前提とする代数的仕様, 電子情報通信学会論文誌 D, Vol.J69-D, No.4, pp. 324–331 (1986).

[9] ル一光, 栗屋英司, 関浩之, 藤井護, 二宮清: 抽象的順序機械の形で記述された代数的仕様からプログラムへの変換について, 電子情報通信学会論文誌 D-I, Vol.J73-D-I, No.2, pp. 201–213 (1990).

[10] 田中哲雄, 谷口健一, 奥井順: スクリーンエディタの代数的仕様記述とその実現, 電子情報通信学会論文誌 D, Vol.J71-D, No.7, pp. 1207–1217 (1988).

[11] 工藤朋之, 石原靖哲, 関浩之, 奥井順: 抽象的順序機械型代数的仕様からのドキュメント作成システム, 情報処理学会論文誌, Vol.38, No.7, pp. 1412–1424 (1997).

[12] 辻野嘉宏: GUIダイアログのための検証法について, 電子情報通信学会論文誌 D-I, Vol.J82-D-I, No.10, pp. 1286–1294 (1999).

[13] Moher, T., Dirda, V., Bastide, R. and Palanque, P.: Monolingual, Articulated Modeling of Users, Devices, and Interfaces, *Design, Specification and Verification of Interactive Systems'96*, Springer-Verlag, pp. 312–329 (1996).

[14] Clark, R. G. and Moreira, A. M. D.: Formal Specification of User Requirements, *Automated Software Engineering 6*, Kluwer Academic Publishers, pp. 217–232 (1999).

[15] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W.: *Object-Oriented Programming and Design*, Prentice-Hall (1991).

[16] Cabrera, M., Torres, J. C. and Gea, M.: Towards User Interfaces Prototyping from Algebraic Specification, *Design, Specification and Verification of Interactive Systems'99*, Springer-Verlag, pp. 67–83 (1999).

[17] d'Ausbourg, B. and Cazin, J.: Using TRIO Specifications to Generate Test Cases for an Interactive System, *Design, Specification and Verification of Interactive Systems'99*, Springer-Verlag, pp. 148–166 (1999).

[18] Butterworth, R., Blandford, A. and Duke, D.: The Role of Formal Proof in Modeling Interactive Behavior, *Design, Specification and Verification of Interactive Systems'98*, Springer-Verlag, pp. 87–101 (1998).

[19] Navarre, D., Palanque, P., Bastide, R. and Sy, O.: Structuring Interactive Systems Specifications for Executability and Prototypability, *Design, Specification and Verification of Interactive Systems 2000*, Lecture Notes in Computer Science, Vol. 1946, Springer-Verlag, pp. 97–119 (2000).

[20] Dix, A. J., Finlay, J. E., Abowd, G. D. and Beale, R.: *Human-Computer Interaction*, Prentice Hall Europe, 2nd edition (1998).

[21] Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S. and Carey, T.: *Human-Computer Interaction*, Addison-Wesley (1994).

[22] Shepherd, A.: Analysis and Training in Information Technology Tasks, *Task Analysis for Human-Computer Interaction* (Diaper, D.(ed.)), Ellis Horwood, pp. 15–55 (1989).

[23] Naur, P. et al.: Revised Report on the Algorithmic Language ALGOL 60, *Comm. ACM*, Vol. 6, No. 1, pp. 1–17 (1963).

[24] Clarke, E. M., Grumberg, O. and Peled, D. A.: *Model Checking*, The MIT Press (1999).

[25] Fernandez, J. C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L. and Sighireanu, M.: CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox, *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, Vol. 1102, Springer-Verlag, pp. 437–440 (1996).

[26] Cleaveland, R. and Sims, S.: The NCSU Concurrency Workbench, *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, Vol. 1102, Springer-Verlag, pp. 437–440 (1996).

[27] CWB-NC group at Stony Brook: The Concurrency Workbench of New Century. http://www.cs.sunysb.edu/~cwb/.

[28] ISO: Information technology – Enhancements to LOTOS(E-LOTOS), Final Draft International Standard ISO/IEC FDIS 15437 (2001).

[29] 地平稔: ユーザタスクの形式的記述に基づくインタラクティブシステム設計法, 修士論文, 奈良先端科学技術大学院大学, NAIST-IS-MT9851049 (2000).

[30] Flanagan, D.(ed.): *X toolkit intrinsics reference manual*, O'Reilly, third edition (1992).

[31] Walrath, K. and Campione, M.: *The JFC Swing tutorial*, Addison-Wesley (1999).

[32] Ehrig, H. and Mahr, B.: *Fundamentals of Algebraic Specification 1*, Springer Verlag (1985).

[33] Baader, F. and Nipkow, T.: *Term Rewriting and All That*, Cambridge University Press (1998).

[34] Arnold, K. and Gosling, J.: *The Java Programming Language*, Addison-Wesley, second edition (1998).

[35] Appel, A. W.: *Modern Compiler Implementation in Java*, Cambridge University Press (1998).

研究業績

1．学術専門誌

1-1 池田瑞穂, 高田喜朗, 関浩之: インタラクティブシステム設計法におけるタスク図の形式的定義と形式的検証への応用, コンピュータソフトウェア，日本ソフトウェア科学会，採録決定.

2．国際会議論文集

2-1 Ikeda, M., Takata, Y. and Seki, H.: Formal Specification and Implementation Using Task Flow Diagram in Interactive System Design, *Proceedings of 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001)*, vol.I Information Systems Development, pp.422-428 (2001).

3．研究会発表

3-1 池田瑞穂, 高田喜朗, 関浩之: ユーザタスクの形式的記述に基づくインタラクティブシステム設計法の提案, 情報処理学会研究報告，99-SE-122，pp.25-32 (1999).

3-2 地平稔, 池田瑞穂, 高田喜朗, 関浩之: インタラクティブシステムの設計におけるタスクの形式的記述とその実現, 情報処理学会研究報告，2000-SE-125，pp.59-66 (2000).

3-3 池田瑞穂, 高田喜朗, 関浩之: インタラクティブシステム設計法におけるタスクモデルの形式的記述と検証, 電子情報通信学会技術研究報告，SS2001-12，pp.1-8 (2001).

3-4 池田瑞穂, 高田喜朗, 関浩之: ユーザインタフェースの代数的仕様と仕様からのプログラム生成, 情報処理学会研究報告，2001-SE-135，pp.9-16 (2001).

4．全国大会発表

4-1 池田瑞穂, 高田喜朗, 関浩之: インタラクティブシステム設計におけるタスクの形式的記述とその実現, 情報処理学会第 58 回全国大会，講演番号 2C-04 (1999).