B Method における
部品再利用によるソフトウェア合成と
高信頼ソフトウェア部品の整備

中村 丈洋

電気通信大学大学院 電気通信学研究科

博士(工学) 学位申請論文

2014年 3月

B Method における
部品再利用によるソフトウェア合成と
高信頼ソフトウェア部品の整備

博士論文審査委員会

主査	西野　哲朗	教授
委員	渡辺　成良	名誉教授
委員	高橋　治久	教授
委員	柏原　昭博	教授
委員	寺田　実	准教授
委員	庄野　逸	准教授

著作権所有者
中村 丈洋 2014年

Abstract

We propose 'software development method' based on software generation. In this method, we get dependable software easy based on formal method. And also, we propose automatic dependable software components generation, and help us to apply software synthesis.

In recent years, cost and reliability of software is big thema in software development. 'How to develop reliable software easy' is one of the big goal of software engineering. An drastic approach for this goal is software synthesis, which generate software without human conding. However, software syntehsis needs software components for each software domain, and the reliability of software is depends on one of components.

In thsi research, we propose 'Model Satisfiable Software Synthesis (MSSS)'. In this method, we define 'Model Satisfiable Fine-grained Component (MSFC)' based on B method, and also define reliability of software synthesis and slicing method. Our method is based on B method and it's use set theorem and predicate theorem to describe software. So MSSS method can be applied for software domain written with this theorem, it's means that we can't apply our method for UI and asynchronous system.

We proved MSFC generation method itself based on its mathematical definition, and MSSS can synthesis reliable software because of mathematical software search and reuse. However, the software search based on mathematics is costly to check reusable between requirement and software components. So we provide software model normalization method, which make same text between mathematical equal models.

In this research we apply our method for some little systems, e.g. bank system, rental-car system. In these examples, we prove generated software and software components with theorem prover of B method tool. However, there is not enough public practical case study of B. and we need more practical examples to refine our method.

MSSS method will enable us to apply synthesis to more software domains, and to develop dependable software quickly in lower cost. Also it will release us from routine work and assign to more creative works.

概要

本研究では信頼性を定理証明により定性的に評価可能な高信頼ソフトウェアを形式仕様から自動生成する手法の提案を目的とする．また，この手法の実現にあたり課題となる，高信頼部品整備を容易にするため，既存の高信頼ソフトウェアからソフトウェア部品群を自動生成する手法を提案する．

近年のソフトウェアの大規模複雑化に伴い，開発コストの増大と信頼性の低下が問題となっている．高信頼なソフトウェアを容易に開発することはソフトウェア工学の大目標の一つといえる．これに対するドラスティックなアプローチとしては入力仕様や設計から人手によらずコードを生成する自動コード生成が挙げられる．自動コード生成は開発コストの低減や開発期間の短縮だけでなく，人による誤りが混入せず信頼性向上にも寄与する．一方で，生成ソフトウェアの信頼性が部品の信頼性に依存し，高信頼なソフトウェア部品を開発対象ごとに整備する必要があり，このコストが手法適用の妨げとなる．

本研究では数学を基盤としてソフトウェアの信頼性を定性的に保証する形式手法 B Method の枠組みをソフトウェア部品と再利用の自動化に応用することで，高信頼ソフトウェア部品の自動生成とその再利用による新たなソフトウェア開発手法「モデル充足ソフトウェア合成（MSSS）手法」を提案する．B Method は J.R.Abrial らにより提案された形式手法であり，大きな特徴として数学的仕様と命令型言語による実装間の整合性を定理証明に保証できる点である．MSSS 手法ではこの B Method の信頼性保証の枠組みを応用してモデル充足ソフトウェア部品 (MSFC) を定義する．

MSSS 手法は基盤とする数学的仕様の性質上，集合論と述語論理で記述できる範囲内でしかソフトウェアの仕様を記述できず，非同期処理やユーザインタフェースは自動合成の対象外となる．一方で，ソフトウェアの信頼性と MSFC の信頼性を数学的に定義することで，MSSS 手法では部品生成手法とソフトウェア合成手法の信頼性を数学的に定義でき，定性的な評価が可能である．本研究ではソフトウェア部品の生成手法自体に定理証明を適用し，その信頼性を定性的に保証する事を試みた．これにより，細分化モデルの信頼性を保証するのに必要な推論器の性質と，制約条件の抽出条件を定理証明により得られた．ソフトウェア合成についても提案した合成手順で部品の実装間の制約条件の矛盾以外は保証できることを示し，合成結果に矛盾が生じた際の低コストな解決手段を提案した．

MSSS 手法のように数学的判定を必要とする部品自動再利用では膨大な部品群に対して数学的判定を行うため，計算コストの低減が問題となる．本研究ではこの問題に対して，数学的に意味の等しい数学的仕様の字面が一致し，また，含意関係となる字面が完全部分一致となるようモデル細分化手法を提案した．これにより，文字列一致による効率的な部品検索を可能にした．

本研究では MSSS 手法の適用例として銀行口座システムなどに対する MSFC 生成とレンタカーシステムなどの自動合成を行った．これにより生成された MSFC やソフトウェアに対して B Method の証明器を適用し，定義どおりの信頼性が得られることを確認した．ただし，B Method により記述されたソフトウェアは現状では広く公開されておらず，より実践的な手法の適用が今後の課題となる．

近年では，システムの不具合が莫大な賠償や会社の信用問題に発展するケースが相次いでおり，一般企業にも高信頼なシステム開発が求められている．しかし，高信頼ソフトウェア開発手法はその高い開発コストゆえに普及していないのが実情である．このため，MSSS 手法により高信頼ソフトウェア開発を自動化することは，高信頼ソフトウェア開発の導入コストを低減し，それを普及する為にも重要であると考える．また，高信頼ソフトウェア開発の自動化により人間はデバッグやコーディング作業から解放され，より上流工程の創造的作業に専念できると期待できる．

目 次

図 目 次

表 目 次

第1章

はじめに

近年のソフトウェアの大規模複雑化に伴い開発コストの増大と信頼性の低下が問題となっている．高信頼なソフトウェアを容易に開発することはソフトウェア工学の目的の原点といえる．この目的に対するドラスティックなアプローチとして自動コード生成が挙げられる．自動コード生成は入力された仕様や設計に対して実装を人手によらず自動生成する手法であり，開発期間の短縮だけでなく，人間によるコーディングミスが生じないため信頼性向上に寄与する [15].

人間が扱いやすい記述から実行可能なコードを生成する点で自動コード生成は C 言語などを機械語に翻訳するコンパイラと類似が認められる．自動コード生成のコンパイラとの違いとして‘アルゴリズムが明記されない入力からアルゴリズムを含む実装を生成する’ 点が挙げられる．詳細は 2.3 節で説明するが，本研究ではアルゴリズムを含む実装の生成方法により既存の自動コード生成手法を大きく‘定理証明に基づく手法’ と‘経験則に基づく手法’ に分類する．‘定理証明に基づく手法’ は入出力のデータ間関係を数学的に記述した仕様からそれを実現するアルゴリズムを定理証明に基づき生成する [3, 27]．これに対して，経験則に基づく手法は仕様や実装に対するアルゴリズムのパターンや部品を整備し，それを再利用することでアルゴリズムを含んだコードを生成する [36, 39]．これらの手法は表 1.1 に示すようにそれぞれ長所と短所がある．本稿で提案する‘モデル充足ソフトウェア合成 (MSSS) フレームワーク’ は以下の着眼点により既存合成手法の問題点克服をはかる．

1. 高信頼ソフトウェア開発手法である B Method を応用したソフトウェア部品の信頼性保証

表 1.1: 自動コード生成手法の特徴比較

	MSSS フレームワーク (提案手法)	定理証明に基づく手法	経験則に基づく手法
信頼性	定理証明に劣るが高信頼	高信頼	静的保証なし
適用性	部品の整備が容易	低い	部品の整備が困難
計算量	ダイジェストや字面一致で低減	計算困難	実用的

2. 数学的な要求仕様と部品の仕様間のマッチングによる高信頼なソフトウェア部品再利用
3. B Method で開発されたソフトウェアからの高信頼なソフトウェア部品の自動生成

定理証明に基づく手法では仕様を満たすアルゴリズムの生成に定理証明を用いたため，実用的な計算量を実現できなかった．これに対して MSSS フレームワークでは信頼性の保証に定理証明を用いるがアルゴリズムを部品として再利用する事で実用的な計算量を実現する．また，経験則に基づく手法の信頼性の問題点を着眼点 (1), (2) により解決する．部品再利用に基づく合成手法では部品や再利用手法の信頼性が合成ソフトウェアの信頼性を左右する．経験則に基づく手法ではこれらの信頼性を経験的に保証するが，提案手法では高信頼な部品群に数学に基づく健全な再利用を適用することで部品と合成手法の信頼性を静的に保証する．また，経験則に基づく手法の適用性の問題点を着眼点 (3) により解決する．一般的に部品再利用に基づく合成手法では問題領域毎に部品を整備する必要があり，部品整備が容易でないことがソフトウェア合成手法の適用性を低下させていた．これに対して，提案手法は既存ソフトウェアから高信頼な部品を自動生成することで部品整備に要する作業量を軽減し，合成手法の適用性を向上させる．これにより MSSS フレームワークは表 1.1 のように定理証明に基づく手法のように高信頼でありながら，経験則に基づく手法のように実用的な計算量をもち，さらに部品の整備が容易な事で高い適用性を持つ．

この様に提案手法は既存のコード生成手法に比べて多くの長所を持つが，それ故にその実現には解決すべき多くの課題を持つ．詳細については手法の詳細説明で行うが，特に注目すべき問題として以下の事が挙げられる．

1. 適用可能な問題領域の特定
2. 自動生成したソフトウェア部品の信頼性保証
3. 合成したソフトウェアの信頼性保証
4. 実用的な計算量でのソフトウェア合成
5. 作業量の軽減

MSSS フレームワークは信頼性保証の枠組みとして B Method を応用するため，(1) について適用可能な問題領域も B Method の枠組みで限定される事は明らかである．本稿ではこれに加えて B Method では適用可能だが MSSS が適用できない問題領域が存在するか，また，そのような問題に対してどの様な対応策があるかを示す．(2) についてソフトウェア部品の信頼性は B Method の枠組みを応用して定義する．さらに，自動生成されたソフトウェア部品の信頼性が保証される条件を信頼性の定義から明らかにする．(3) については MSSS で合成したソフトウェアの信頼性を B Method の枠組みで保証する際に，検証作業なしに何が保証されて，何が保証されないのかを明らかにする．MSSS は定

理証明に基づき信頼性を保証するため，(4) の計算量低減が必要である．本稿では ‘文字列一致による数学的同値判定’ をはじめとした計算量低減を提案する．MSSS の入力は数学的仕様であり従来手法に比べて抽象度が高いため，数学的には等しいが実装が異なる場合に入力だけでは与えきれない要件が生じる．このため，実装の選択等で利用者による作業が生じ，(5) の作業量低減が必要となる．冒頭にも述べたように作業量の軽減は開発コストの低減だけでなく成果物の信頼性にも関わる問題である．本稿では作業量軽減のために部品の読解が不要な実装の選択の提案などを提案する．

MSSS で応用する B Method は高信頼ソフトウェア開発手法としては比較的新しい手法であるが，既にパリメトロ 14 号線の自動運転システムをはじめとして多くの実用事例を持つ手法である．近年ではシステムの不具合が莫大な賠償や会社の信用問題に発展するケースが相次いでおり，一般企業にも高信頼なシステム開発が求められている．しかし，高信頼ソフトウェア開発手法はその高い開発コストゆえに普及していないのが実情である．このため，MSSS フレームワークにより高信頼ソフトウェア開発を自動化することは，高信頼ソフトウェア開発の導入コストを低減し，それを普及する為にも重要である．また，高信頼ソフトウェア開発の自動化により人間はデバッグやコーディング作業から解放され，より上流工程の創造的作業に専念できると期待できる．

第2章

先行研究

本研究はソフトウェア開発の高信頼化とそれに要するコストの低減を形式手法と部品再利用・生成の自動化により低減することを目的とする．本節ではこの研究背景として近年のソフトウェア開発における信頼性検証，ソフトウェア開発の分業化を紹介する．また，本研究を進めるにあたり，関連研究となる自動コード生成，ソフトウェア部品再利用，形式手法 B を紹介する．

2.1 ソフトウェア開発における信頼性検証

ソフトウェア開発は分野によりそれぞれ要求が異なるが，近年において共通して求められている観点として信頼性が挙げられる．しかし，信頼性の向上には相応の開発コストが必要であり，許容できるコストにより信頼性向上のアプローチは異なる．本節では高信頼領域とそれ以外の領域において，現在どの様な信頼性検証手法が用いられているかを紹介する．

2.1.1 形式手法に基づく信頼性検証

信頼性に対してより多くの開発コストが許容される宇宙開発やインフラなどのシステム開発では多重化や形式手法により信頼性向上が図られてきた．特に形式手法は非属人的なソフトウェアの信頼性保証が可能であり，高い信頼性が求められるシステム開発での実用事例が数多く報告されている [32]．これらの事例の中で特に注目すべき点として，高信頼システム開発分野に限定されるものの，形式手法が商業的にも成功を収めつつある点である．証明により仕様とソフトウェアの信頼性を保証する形式手法 B Method を用いて開発された鉄道システムがフランスで開発されたが，このシステムは欧州各国だけでなく，北米，南米，東アジアの各国に輸出されている．一方で，日本国内でも FeliCa ファームウェア開発のような形式手法の適用事例は存在するものの，証明ベースの厳格な形式手法は欧州に比べて特筆できる成功事例が無いのが実情である．

2.1.2　テスト駆動による信頼性検証

より小規模，あるいは信頼性に妥協が許されるソフトウェア開発ではソフトウェアテスト技術の研究と応用が盛んである．この中でも近年注目される技術としてテスト駆動開発 (TDD) とそこから派生した振舞駆動開発 (BDD) が挙げられる．テスト駆動開発はコーディング前にコードを検査するためのテストを記述し，このテスト結果が真になるよう，開発を進めるソフトウェア開発手法である [10]．このテストを開発工程の随所で実行する事で品質改善を行う開発手法として継続的インテグレーション (CI) が挙げられる．近代的なプログラミング言語では xUnit[18] などのテスト記述フレームワークが提供されており，これらのフレームワークとテスト実行を管理するツール [17] を用いて CI を非属人化する事が可能である．TDD ではテスト記述者が機能定義を読解し，これを満たすテストを記述する．この人手による機能定義の読解を形式的に記述したソフトウェア機能定義書により自動化した手法として振舞駆動開発 (BDD) がある．この手法で記述される機能定義書は証明ベースの厳格な形式手法と同様に集合論と述語論理で記述される．この両者で大きく異なる点としては BDD では証明ベースではなく，テストケースの自動生成により検証を行う点が挙げられる．このため，証明ベースの手法では本来偽になる検証結果を真であると誤判定する事は無いが，BDD ではこのような誤判定が生じうる．一方で，CI のように繰り返し検証を行う環境ではコード変更毎に証明を行うことは困難であり，BDD のようにテストケースを自動生成する手法との相性がよい．

以上のように，高信頼ソフトウェア開発の領域では証明ベースの厳格な形式手法が，より小規模な開発では CI ツールを用いた TDD や BDD が実践されている．この動向のなかで，我々が注目している点は，これまで一般的なソフトウェア開発では倦厭されてきた形式仕様が，CI への注目により BDD の振舞仕様として記述されている点である．本研究は高信頼ソフトウェア開発領域で培われた証明ベースの開発手法を一般的なソフトウェア開発にも応用することを目的としており，その前提として形式的な仕様記述が必要となる．この前提は近年の BDD への注目から想定可能であると考えている．

2.2　ソフトウェア開発の分業化

大規模なシステム開発では開発工程を並列化できるように分業化することで大量のリソース投入による短期間での開発を可能にしている．特に近年は要求の高度化やオフショアによる開発コスト低減のために，中小規模な開発においても開発を分業化する必要がある．最も単純な開発工程の分業化はウォーターフォールにおける各工程の分離であるが，ここでは，特に設計・実装工程における分業化に注目し，MVC アーキテクチャ，ソフトウェアプロダクトラインを紹介する．

2.2.1　MVC アーキテクチャ

ソフトウェアの機能間の独立性はソフトウェア開発の黎明期からの課題であり，プログラミングの基本として大域変数への配慮，ソフトウェアのモジュール化などはその基本として挙げられる．特に GUI が一般的になった現在においてはより洗練された UI を提供するために，GUI の開発をいかに独立させ，かつ，ロジックやデータと連携させるかが課題となった．この課題に対する代表的な手法として MVC アーキテクチャが挙げられる．MVC アーキテクチャはデータ表現 (Model), ユーザインタフェース (View), 制御 (Controller) にソフトウェアの役割を分割し，連携させることで開発者の責務を明確にする．MVC において興味深い点として，MVC による多くのソフトウェア開発では View の記述言語は Controller の記述言語と異なる点である．例えば，Java による Web アプリケーションでは JSP などのマークアップ言語を用いて View を記述する．また，多くの開発環境では GUI をコーディングによらずグラフィカルに設計できるツールが提供されておりコードとは別に管理される．本研究で提案するソフトウェア合成手法は MVC で言えばデータ表現と制御にのみ適用可能であり View には適用できない．しかし，上述の MVC の観点から言えば，Model と Controller の記述に適していれば View に適用できない事自体は問題とならないと考えられる．

2.2.2　ソフトウェアプロダクトライン

ソフトウェアプロダクトラインとはプログラムやシステムの部品，それらの関連性，システムデザインと拡張の統制を構造化した開発方式である [12]．この開発方式の中ではソフトウェア開発者は主にプロダクトラインマネージャ，プロダクトラインアナライザ，部品開発者，部品組立者，ブローカーという役割に分類される．プロダクトラインマネージャは開発計画を立て，開発組織全体を管理する．プロダクトラインアナライザは開発成果物を精査し，開発計画に合わせてシステム構成や部品化の方策を決定する．部品開発者は部品化方策に合わせてソフトウェア部品の仕様を明示し，部品を開発，保守する．部品組立者は提供された部品を組み立てることでソフトウェアを開発する．ブローカーは開発された部品を部品ライブラリに編纂し，開発チーム間での部品共有をサポートする．このような開発組織の分業化は携帯電話などの類似仕様で繰り返し開発が行われる分野において特に有効である [9]．また，近年では部品ライブラリの提供者が数多く存在し，この様な分業によるソフトウェア開発は特に Web アプリケーションや携帯端末アプリケーションの開発において一般的に行われている．

一方で，ソフトウェアの大規模化や部品ライブラリの肥大化に伴い，部品の重複を避けて部品化を行う事や，大量の部品群からソフトウェアを再利用することが困難になりつつある．本研究では部品化と部品組立を自動化することにより，これらの問題への対

処を考える．

2.3　自動コード生成

自動コード生成はアルゴリズムを含まないソフトウェアの仕様書や設計書から実行可能なコードやコードの雛形を自動生成する手法である．本稿ではアルゴリズムの注入方法から自動コード生成を ‘経験則に基づく自動コード生成’ と ‘定理証明に基づく自動コード生成’ に大分した．以下の節ではそれぞれの手法について説明する．

2.3.1　経験則に基づく自動コード生成

ソフトウェア合成は人手によらずソフトウェアを開発することで開発期間の短縮と人間による誤り混入を低減する [15]．ソフトウェア合成はアルゴリズムなどを含まない抽象的な仕様や設計からソフトウェアを生成する自動化手法であり，MDA などが普及しつつある [13, 5]．MDA では ‘オブジェクト指向に基づく再利用’ と ‘ドメイン特化言語 (DSL) に基づくコード記述’ によりソフトウェアを合成する [36]．DSL は実行可能コードに翻訳可能な文法や単語を問題領域毎に定めた言語で，ソフトウェアを自然言語に近い表現で抽象的に記述できる．例えば，アカウント管理ならば，‘regist A as B’ という文法と ‘Manager’ という単語を定め，‘regist userA as Manager’ というコードで userA を Manager という役割で登録することが考えられる．これにより，データベースなどの実装方法は隠蔽される．DSL の単語や文法は抽象的な記述で実装を得られる点で，オブジェクト指向におけるクラスと同様にソフトウェア部品と捉えることができ，MDA はソフトウェア部品再利用の自動化と捉えられる．ソフトウェア合成を目的とした部品再利用では ‘実装の抽象化’ と ‘再利用の健全性’ が重要な観点となる．実装の抽象化は部品の実装をクラス名や関数名などにより隠蔽することであり，合成の入力となる要求仕様の抽象化に必要である．また，再利用の健全性は合成ソフトウェアが要求を満たすために必要であり，要求に合致しない部品を再利用しないこと，部品を矛盾無く結合できることが求められる．このようにソフトウェア合成を部品再利用と捉えると以下の課題が挙げられる．

1. リリース直後の部品の信頼性保証が不十分である．
2. 大量の部品の名前とそれにより抽象化される実装を把握する必要がある．
3. 部品整備のコストが大きく適用可能な問題領域が限定される．

課題 1 は部品の信頼性を再利用の実績から評価するために生じる．この課題に対するアプローチとして定理証明によるパターンの信頼性保証 [4] が挙げられる．この手法は部品に数学的な仕様を与え，実装が数学的な仕様を満たすことを定理証明によって保証する手法である．ただし，数学的な仕様記述はあいまいさを排除するために自然言語によ

る仕様記述に比べて厳密さを求められ，膨大な量の部品群に対して数学的な仕様を付加することは容易ではない．また，定理証明による信頼性保証は大きな労力と計算量を必要とするため，課題 (3) への対処が必要となる．課題 2 は MDA がクラス名や DSL の語彙などの名前により実装を抽象化するために生じ，部品を健全に再利用するには名前と実装の把握が必要となる．DSL は問題領域に特化することでこの問題の解決を図っており，良く設計された DSL は語彙が小さく，また，対象の問題領域の知識に基づき語彙を構成するため，語彙の把握が容易である．課題 3 はソフトウェア合成には大量の部品群が必要であり，それらの多くが問題領域に依存することで生じる．また，名前により実装を抽象化する場合，語彙が異なる問題領域間で部品を共有できない事も問題となる．この課題の改善策として部品の低機能化と一般化が考えられる．例えば，データベースライブラリを容易に扱うために SQL に似せた DSL などが実在する．しかし，この様な DSL により課題 3 の解決を図ると膨大なライブラリに対応する DSL を把握する必要が生じる．この様に，既存のソフトウェア合成では課題 3 と課題 2 がトレードオフの関係にある．本研究では部品整備の問題を ‘部品が存在しない問題領域における部品整備の自動化’ と ‘不足部品の生成補助’ という観点から議論する．

本稿で提案する自動コード生成手法は後述する B Method という形式手法の記法を用いて要求仕様を記述する．この，B Method のツールには ‘B Automatic Refinement Tool (BART)’ という自動化ツールが存在する．BART は要求仕様を集合論と述語論理で記述するため抽象度が高く，名前に頼らないパターンマッチが可能である．また，BART 自体は仕様と実装間の信頼性保証を考慮していないが B Method は仕様と実装間の信頼性保証の枠組みを持つため，これを応用することで BART のパターンの信頼性を保証できると考えられる．このため，BART は経験則に基づく自動コード生成の課題 (1) と課題 (2) に対して大きな改善が見られる．しかし，先に述べたように実装への数学的仕様の付加と仕様と実装間の信頼性保証は大きな労力を要するため，課題 (3) に対する対処が必要となる．

2.3.2　定理証明に基づく自動コード生成

入出力のデータ間関係を数学的に記述した仕様からそれを満たすコードを定理証明に基づき出力する手法 [3, 27] を本稿では定理証明に基づく手法と呼ぶ．定理証明に基づく手法は人工知能の分野で盛んに研究されてきた経緯があり，現在においても定理証明に基づくコード生成は人工知能や計算機科学の分野で議論されているが，ソフトウェア工学において十分な議論がなされているとは言えない．本稿で提案する自動コード合成手法はその信頼性保証に定理証明を用いるため，本節では定理証明に基づく手法の概要と本稿で提案する手法との関連を説明する．

定理証明に基づく手法にはいくつか種類があるが, その一つに一階述語論理で記述され

た仕様から prolog の様な実行可能な一階述語論理のサブセットを演繹的に推論する手法がある．この手法は信頼性が高く, また, 先に紹介した経験則に基づく手法と異なり推論により実装に必要な知識を導出するため知識の蓄積が必要ない．しかし, 定理証明という困難な計算を必要とし, 定理証明で推論可能な知識でしか実装を出力できないため適用可能なソフトウェア領域が限定される．このため, 定理証明に基づく手法は経験則に基づく手法に比べて開発現場への応用に乏しい．本稿で提案する自動コード生成手法では集合論と述語論理でソフトウェアの仕様を記述するが，同様の記述を入力とする定理証明に基づく手法に KIDS[22] がある．KIDS では問題領域毎にその問題の推論に用いられる領域理論を (Domain Theory) を整備し，仕様で与えられたソフトウェアの入出力間関係が真であることを演繹的に推論する事で LISP のコードを生成する．

本稿で提案する自動コード合成手法と KIDS をはじめとする定理証明に基づく手法は必ずしも競合しない．これは，定理証明に基づく手法が対象とする問題の粒度が極めて小さい事による．例えば，KIDS では n 女王問題を解くアルゴリズムを自動生成できるが，本稿で提案する手法では n 女王問題が部品の粒度であるため，‘n 女王問題を解く部品’ を与える必要がある．また，本稿で提案する手法で用いる部品は仕様と実装間の信頼性保証に定理証明を用いる．定理証明に基づくコード生成と定理証明による信頼性保証では一般的に信頼性保証の方が容易である．これは，定理証明に基づくコード生成手法は広大な空間から解となる領域理論の組み合わせを推論により探索することでコードを得るのに対して，定理証明による信頼性保証はコードを変換した命題群と基底となる命題群の間を結ぶ推論経路が存在することを示せば良いためである．

2.4　ソフトウェア部品再利用

ソフトウェア部品再利用はソフトウェア開発黎明期から今日までソフトウェア開発コストの低減と信頼性向上に大きく貢献してきた．ソフトウェア部品の例としてはオブジェクト指向言語における JAVA API ライブラリや Standard Template Library が挙げられる．また，構造化手法における再利用可能な関数や手続きもソフトウェア部品になりうる．ここで，関数や手続きはクラスよりも小さな機能体であるが，このことを関数や手続きは部品としてみたときクラスよりも ‘細粒度’ であると言う．逆に部品が大きな機能体である場合には ‘粗粒度’ であると言う．この様にソフトウェア部品は用途に応じて粒度が大きく異なる．

ソフトウェア部品の重要な性質として汎用性と機能性が挙げられる．汎用性は 1 つの部品をどれだけ多くの場面で再利用できるかを表し，機能性は 1 つの部品を再利用することでどれだけの機能が得られるかを表す．本研究では汎用性と機能性をまとめて再利用性と呼ぶ．また，ソフトウェア部品を再利用するためには部品が提供する機能や利用可能な場面を利用者や計算機が把握する必要がある．この様な情報の例としては関数名

や部品名，自然言語による仕様などが挙げられる．

2.4.1 項では細粒度ソフトウェア部品について説明し，また，2.4.2 項ではソフトウェアの細粒度化に用いられるプログラムスライシングを紹介する．2.4.3 項では部品に対する仕様の付加について説明する．

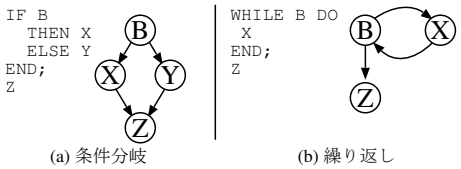
2.4.1　細粒度ソフトウェア部品リポジトリ

細粒度ソフトウェア部品 (fine-graned software component) は一般的にはモジュールよりも粒度が細かい部品であり，本稿ではさらに意味を限定して関数や手続きよりも粒度が細かい部品を細粒度ソフトウェア部品と呼ぶ．阿草らは細粒度リポジトリにおける CASE ツール プラットフォーム Sapid を提案した [35]．CASE とは Computer Aided Software Engineering の略であり，ソフトウェア開発やその保守に計算機を応用することであり，CASE ツールはそれに用いられる技術やソフトウェアである．従来の CASE ツールがモジュールをソフトウェアの構成単位として解析を行うのに対して，Sapid はソフトウェアをモジュールより細かな単位に細分化して細粒度リポジトリに集積することで，より細かい粒度でコードの解析を可能にした．Sapid の特徴的な点としてソフトウェアの細分化を 2.4.2 項のプログラムスライシングを応用して機械的に行う事が挙げられる．阿草らはさらに細粒度リポジトリに集積されるコードの部品化，再利用についても言及しており，それにより以下のような利点が生じると述べている [26]．

1. 部品管理資源の削減
2. メンテナンスの簡素化
3. 部品の信頼性の向上

利点 (1) における部品管理資源とは部品を格納するのに必要なデータベースの容量などをいい，細粒度化により各部品間でコードの重複が少なくなるため保存に必要な容量が抑えられる．利点 (2) はコードの重複が少ないため，同じ修正や変更を各部品に行う必要がなく，メンテナンスが容易になるためである．利点 (3) は細粒度化により各部品の再利用回数が増えることにより，部品の不具合やアルゴリズムが修正され，部品の信頼性が向上するためである．

上記の様に細粒度部品リポジトリは部品の粒度が細かいことにより高い汎用性を持つが，機能性が低く，1 つの部品を再利用しても得られる機能が少ない．この様に，再利用性を粒度の点から見ると一般的に汎用性と機能性はトレードオフの関係にあり，細粒度ソフトウェア部品は汎用性は高いが機能性は低い．そのため，細粒度部品を扱う際には部品再利用を容易にする仕組みが必要となる．



(a) 条件分岐　(b) 繰り返し

図 2.1: IF 文のグラフ表現

2.4.2　プログラムスライシング

プログラムスライシングはプログラムを要約する手法の一つであり，プログラム中で興味がある文の働きを理解しやすいように理解に不要な部分を取り除く．プログラムスライシングは元々はプログラムの保守に用いられてきたが，先に紹介した Sapid ではプログラムスライシングの代表的手法である Weiser の手法 [24] を細粒度リポジトリの構築に利用している．本稿で紹介する提案手法でもソフトウェア部品の生成にプログラムスライシングを応用する．

Weiser の手法は命令型言語においてプログラム P とそのプログラム中のある行 i と変数群 V を入力として行 i における変数群 V の状態を決定するのに必要十分なプログラム S を出力する．Weiser の手法ではプログラムの各行をノード，プログラムの流れをエッジとし，プログラム P を有効グラフとして扱う．例えば図 2.1(a) の様に IF B THEN X ELSE Y END Z からなる条件分岐は B, X, Y, Z の 4 ノードからなり，(B, X), (B, Y), (X, Z), (Y, Z) の 4 つのエッジを持つ．また，図 2.1(b) の様に WHILE B DO X END; Z からなる繰り返しは B, X, Z の 3 ノードからなり，(B, X), (X, B), (B, Z) の 3 つのエッジを持つ．

Weiser の手法はノード i, 変数群 V に対してスライシングの基準 $C = (i, V)$ を定め，ノード i における変数群 V の状態を決定するのに必要十分なグラフ S_C を生成する．以下に Weiser の手法の要点を抜粋する．以下のように DEF, REF, $INFL$, BC を定義する．

$DEF(n)$: ノード n において定義される変数群

$REF(n)$: ノード n において参照される変数群

$INFL(b)$: *IF* 文や *WHILE* 文などの分岐であるノード b においてその分岐の内部で実行されるノードの集合

$BC(b)$: $(b, REF(b))$ で表される分岐 b についてのスライシングの基準

この時，以下のように再帰的に S_C^i を求め，収束した値がスライシングの結果である．

1. ノード n の変数群のうち，基準 C において直接影響する変数群を $R_C^0(n)$ とする．$C = (i, V)$ としたとき，$R_C^0(n)$ は以下の (a)，(b) を満たす全ての変数 v の集合である．

(a) $n = i$ であり，$v \in V$

(b) 以下のいずれかを満たすようなノード m に対してエッジ (n, m) が存在する．

i. $v \in REF(n)$ かつ，$w \in DEF(n) \wedge w \in R^0_C(m)$ を満たす変数 w が存在する．

ii. $v \notin DEF(n)$ かつ，$v \in R^0_C(m)$ である．

2. 基準 C に対して直接影響する R^0_C についてのスライシング S^0_C を $S^0_C = \{n \mid R^0_C(n) \cap DEF(n) \neq \emptyset\}$ と定義する．
3. S^0_C が含むノードの実行を制御する条件分岐の集合 B^0_C を $B^0_C = \{b \mid \exists n \in S^0_C \wedge n \in INFL(b)\}$ と定義する．
4. R^0_C, S^0_C, B^0_C を初期値として以下の R^i_C, S^i_C, B^i_C を収束するまで再帰的に計算する．
   $R^{i+1}_C(n) = R^i_C \bigcup_{b \in B^i_C} R^0_{BC(b)}(n)$
   $S^{i+1}_C = \{n \mid n \in B^i_C \vee R^{i+1}_C(n) \cap DEF(n) \neq \emptyset\}$
   $B^{i+1}_C = \bigcup_{n \in S^{i+1}_C} INFL(n)$

2.4.3　ソフトウェア部品への仕様付加

一般にソフトウェア部品を集積した部品リポジトリには部品だけでなく，その部品を説明する文章などが格納される．例えば，JAVA API ライブラリでは各クラスやメソッドの説明が Java Doc として付加されている．Java Doc ではクラスのコンストラクタやメソッドの引数の説明，メソッドの返り値やメソッドで変化するオブジェクトの状態，引数が null である場合の動作などが自然言語で記述されている．これらの文章はライブラリに格納されたクラスの仕様であり，ライブラリの利用者はこの仕様を正しく解釈することでクラスを正しく利用できる．ソフトウェア部品が高信頼であるためには ‘部品が仕様を持つこと’ と ‘品質が保証されていること’ が必要であるとされている [16]．このため，部品に仕様を付加するにあたっては以下を考慮する必要がある．

1. 仕様が一意に解釈できること
2. 部品が仕様を満たすこと

(1) については自然言語では単語が複数の意味を持っていたり修飾語句の係り受けに曖昧さが生じるため一意な解釈が困難である．仕様の解釈が一意でなく，部品記述者の解釈と利用者の解釈が異なる場合には部品が正しく再利用されず誤りを生じる．自然言語の仕様において (2) はその部品の利用実績から経験的に評価される．そのため，利用実績のない部品については部品が仕様を満たすことを評価できない．部品が仕様を満たすことを経験則に頼らず静的に評価するには仕様を計算機で扱う必要がある．

(1) の問題点に対して部品に形式仕様を付加することが提案されている [23, 2, 20]．形式仕様は一意な解釈を与えることができ，計算機で扱うことも容易である．また，B Method や VDM などの形式手法は実装が仕様を満たすことを保証する仕組みを持ち，これを応

用した部品と仕様間の整合性保証が提案されている [4].

しかし，形式仕様の付加や仕様と実装間の整合性保証は大きな労力を必要とする．さらに，部品リポジトリに格納される部品数は膨大であるため，部品リポジトリの生成が困難になると予想される．この問題に対して仕様が付加されていない部品に逆エンジニアリングで形式仕様を付加する手法が提案されている [14, 11]．この手法では形式仕様の付加を自動化でき，さらに仕様が実装を満たすことを保証できる．しかし，ソフトウェアを開発する際には必ず仕様を記述するため，それを破棄して実装から逆エンジニアリングを行うことは部品の仕様と人間の想定する仕様との乖離を招く．

阿草らは 2.4.1 項で紹介した Sapid に対して XML による仕様を付加する Dapid を提案した [25, 28]．Dapid は ソフトウェアの仕様を XML で与え，さらに仕様とコードの対応を与えることで細粒度化されたコードに仕様を与える．阿草らはさらにリポジトリに格納された仕様とコードの整合性の検査を提案している [29]．しかし，自然言語を XML でマークアップした仕様では数学に基づく仕様と異なり仕様の無矛盾性や仕様と実装間の整合性の保証が困難であり，Dapid で検証される整合性は仕様や実装を保守した際の入出力型の一致などに限定される．

2.5　B Method

2.5.1　概要

B Method はソフトウェアの仕様記述から実装工程までを網羅する形式手法である．B Method の記述は ‘モデル’, ‘リファインメント’, ‘実装’ に分けられる．モデルはソフトウェアの仕様であり, 集合論と述語論理による制約と実行順序の概念が無い非決定的な代入で記述される．実装は命令型言語同様に実行順序を持ち結果が一意に定まる代入で記述される．B Method では実装がモデルを満たす事を定理証明によって保証する．しかし, モデルは実装手段を考慮しない抽象的な記述であるため, 要求を直接書き下したモデルと実装手段を明示した実装ではソフトウェアの捉え方に大きな差が生じる．これを埋めるのがリファインメントである．モデルの概念を実装に書き下すことが困難である場合に実装を意識したモデルをリファインメントとして記述して, その記述から実装を書き下す事でモデルの概念を実装に書き下すことが容易になる．

2.5.2　記述の構造

B Method のソフトウェアは図 2.2 に示すような ‘モデル’ と ‘実装’ で構成される．B Method にはこれ以外にもリファインメントという記述が存在する．リファインメントは後述の段階的詳細化を行うための記述であるが，図 2.2 のように段階的詳細化を行わずに

モデルから実装を直接記述することもできる．また，3.3.1 項に述べる理由で本稿ではリファインメントを扱わないため，リファインメントの記述の詳細は割愛する．以降の節ではモデルと実装，および段階的詳細化について説明する．

モデル

B Method のモデルは命題と操作の組で表される．図 2.2(a) は registUser, deposit, withdraw という 3 つの操作からなる簡単な銀行システムのモデルである．図 2.2(a) の一行目はモデル名とモデルパラメタを表す．モデルパラメタはモデル内部においては定数のように振る舞う．本稿ではモデルを式 (2.1) のように数式を用いて表現する．式 (2.1) の組は命題 C_A, P_A, I_A, $Q_{A1}, \ldots, Q_{Ak}$, 代入 U_A, $V_{A1}, \ldots, V_{Ak}$ からなる B Method のモデルを表す．

$$(C_A, P_A, I_A, U_A, ((Q_{A1}, V_{A1}), \ldots, (Q_{Ak}, V_{Ak}))) \tag{2.1}$$

ここで, 命題 C_A, P_A, I_A はそれぞれパラメタ制約 (CONSTRAINTS 節), プロパティ制約 (PROPERTIES 節), 不変条件 (INVARIANT 節) である．パラメタ制約はモデルパラメタとして与えられる値の範囲を制約する．プロパティ制約はパラメタとして与えられない定数の値の範囲を制約する．不変条件は変数の型定義やシステムが守るべき変数間関係を制約する．代入 U_A は初期化を表し, 命題と代入の組のリスト $((Q_{A1}, V_{A1}), \ldots, (Q_{Ak}, V_{Ak}))$ は各操作の代入とその事前条件 (PRE 節) を表す．B Method において命題は集合論と述語論理により記述される．図 2.2(a) の C, I, Q がそれぞれパラメタ制約，不変条件，事前条件であり，これらは集合論と述語論理の命題である．例えば，図 2.2(a) の I は数学的に書き直すと式 (2.2) の様になる．式 (2.2) の不変条件からは ‘氏名 (userName) と住所 (userAddr) の両方が重複する顧客 (user) は存在しない’ という条件が読み取れる．

$$\begin{aligned}
&users \subseteq \{0, \ldots, maxUser\} \land \\
&userAddr \in users \rightarrow seq(0 \ldots 255) \land \\
&\forall\, u1, u2(u1 \in users \land u2 \in users \land \neg(u1 = u2) \\
&\Rightarrow \neg(userName(u1) = userName(u2) \land userAddr(u1) = userAddr(u2)))
\end{aligned} \tag{2.2}$$

また，モデルの代入は図 2.2(a) の V に示すように等価代入文 ($X := E$) や ANY 文, IF 文, SELECT 文などで構成され, それぞれの文は同時代入 (||) で結合される．そのため，モデルの操作には代入順序の概念が存在しない．ANY 文, SELECT 文はともに非決定的な代入を行う文である．ANY 文は非決定的選択文と言い, 非決定的な値を生成し, その値を用いた計算を記述する．例えば，図 2.2(a) の V に含まれる ANY 文は $\{0, \ldots, maxUser\}$ に含まれ，$users$ に含まれない値 $newUser$ を生成する．SELECT 文は条件選択であるが, 複数の条件が同時に真になる場合にはそれらに対応する代入のうち 1 つが非決定的に選択される．

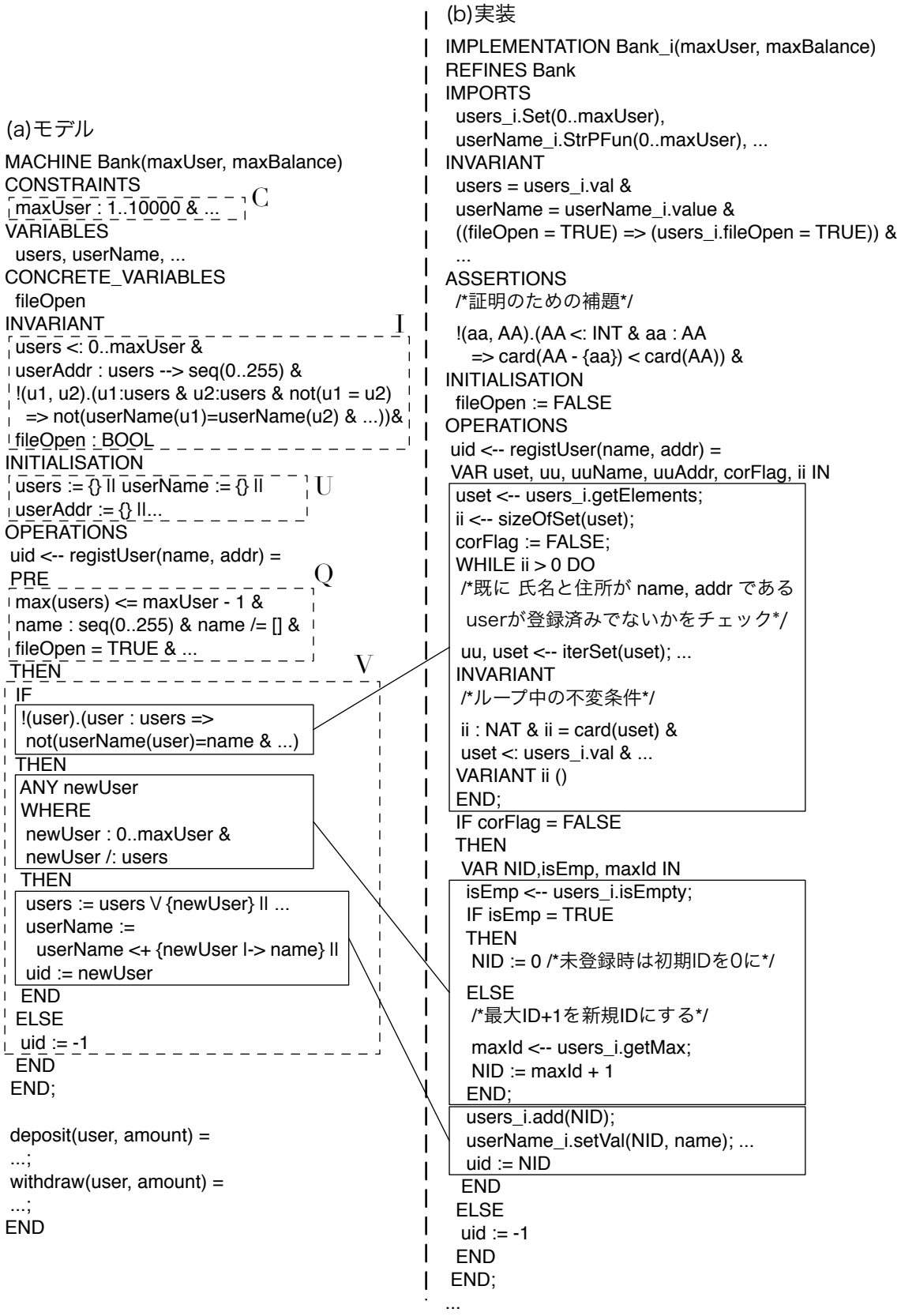


図 2.2: B Method におけるソフトウェアの例

実装

B Method の実装は図 2.2(b) の様に記述される．実装はモデルと同様に命題と操作の組で表現することができ, k 個の操作を持つ実装はプロパティ制約 P_I, 輸入 R_I, 不変条件 I_I, 代入 U_I, $V_{I1}, \ldots, V_{Ik}$ に対して式 (2.3) に示す組で表される．

$$(P_I, R_I, I_I, U_I, (V_{I1}, \ldots, V_{Ik})) \tag{2.3}$$

輸入 R_I は ‘別名. モジュール名’ という書式でモジュールの機能を取り込む記述である．B Method で実装に用いられる基本語彙は極めて小さく, 端末やファイルとの I/O はモジュールを輸入する事で行われる．実装における代入は非決定性を含まない代入文で構成され, それぞれの文は逐次代入 (;) で結合される．図 2.2 において実線で結ばれた記述はそれぞれモデルと実装で対応する記述を表す．モデルの IF 文の条件を計算するために, この例では WHILE ループを利用するが，B Method ではループの停止性保証のために INVARIANT の様な条件を与える．また，モデル中の ANY 文中の非決定的値 newUser に対して登録済 ID の最大値を利用して未登録の値を生成している．図 2.2 中のモデルと実装の対応からも分かるようにモデルによる非決定的な記述は極めて高い表現力を持っており，アルゴリズムを直接記述した実装に比べて高い可読性を持つ．

段階的詳細化

B Method によるソフトウェア開発ではモデルでソフトウェアを抽象的に表現し，そのモデルを満たすよう実行可能な実装を記述する．ここで，モデルは人間が捉えやすいように集合等を定めるのに対して実装は実行効率を考える必要がある．そのため，実装では想定する運用環境で効率よく実行できる内部構造を与える．このように運用環境などの実装依存の記述を加えることを詳細化と呼ぶ．B Method ではモデルを詳細化して直接実装を記述するだけでなく，図 2.3 のように実装を記述しやすいように段階的に詳細化することができる．これを段階的詳細化といい，リファインメントを記述することで行う．

本稿では部品をモデルと実装に似た記述で構成し，この部品に実装非依存な仕様を与える．ここで，部品のモデルを実装非依存な仕様に対するリファインメントとすることが考えられるが，これは不可能である．これは，B Method の詳細化では図 2.3 のように実装依存の制約条件を増やせないためである．すなわち，モデルの操作 V_A の事前条件が Q ならば，実装の操作 V_I は事前条件 Q で実行できなければならない．例えば，V_I が ‘ファイルが開いている’ という実装依存の条件 P を持つならば，モデルの事前条件がこの実装依存の条件 P を持たねばならず，段階的詳細化の過程で条件 P を追加することは出来ない．

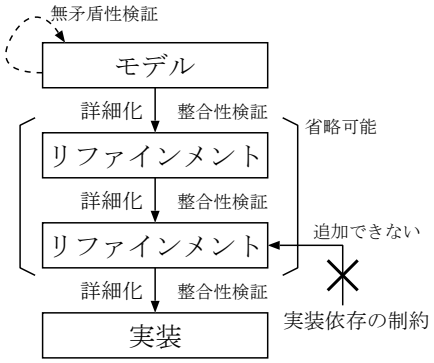


図 2.3: B Method における段階的詳細化

2.5.3　信頼性の定義

B Method ではモデルから実装までの記述が整合性を満たすことを定理証明によって保証する．この定理証明の命題を ‘証明責務’ と呼ぶ．証明責務にはモデルの初期化, モデルの操作, 実装の操作に関する証明責務があり, それぞれ式 (2.4), 式 (2.5), 式 (2.6) の様に表される．B Method では初期化と各操作毎に証明責務が真であることを証明することでソフトウェアの信頼性を保証する．これらの式において代入 V, 命題 I に対して $[V]I$ は V が I を満たすための最弱事前条件といい, V を実行後に I を満たす事を示す命題である．最弱事前条件は右結合であり $[V_{Ij}]\neg([V_{Aj}]\neg I_I)$ における () は省略できる．また，証明責務は $H \Rightarrow G_1 \wedge \cdots \wedge G_n$ の様に含意の後件 (本稿ではゴールと呼ぶ) が論理積で構成される場合，$H \Rightarrow G_1, \ldots, H \Rightarrow G_n$ という命題群を証明することで証明する．この様に証明責務が n 個の命題に分割されるとき，‘本稿では n 個の証明責務が生成される’ と言う．

$$C_A \wedge P_A \Rightarrow [U_A]I_A \tag{2.4}$$

$$C_A \wedge P_A \wedge Q_{Aj} \wedge I_A \Rightarrow [V_{Aj}]I_A \tag{2.5}$$

$$(C_A \wedge P_A \wedge I_A \wedge Q_{Aj}) \wedge (P_I \wedge I_I) \Rightarrow [V_{Ij}]\neg[V_{Aj}]\neg I_I \tag{2.6}$$

モデル M_A と実装 M_I の全ての操作において式 (2.6) の証明責務が真であることを ‘モデル M_A は実装 M_I で詳細化される’ と呼び，$M_A \sqsubseteq M_I$ と表す．この時，実装 M_I が取りうる状態においてモデル M_A の制約条件が真になり，実装 M_I がモデル M_A を満たすことが保証される．

図 2.2 に挙げたモデルと実装の例においてモデル Bank は実装 Bank_i で詳細化される．操作 registUser のモデルの操作から式 (2.5) にしたがってモデルの操作に関する証明責務を生成すると 9 個の証明責務が生成される．また，式 (2.6) にしたがって実装の操作に関する証明責務を生成すると 76 個の証明責務が生成される．式 (2.7) は図 2.2 の実装から生成される証明責務の簡単な例である．式 (2.7) は実装の不変条件から $users = users\_i.val$

であるため真になる.

$$users\_i.val \cup \{max(users\_i.val) + 1\} = users \cup \{max(users\_i.val) + 1\} \tag{2.7}$$

第3章

モデル充足ソフトウェア合成フレームワーク

3.1 背景

2.3節に示したように現在実用化されている既存の自動コード生成手法は生成コードの信頼性保証が不十分である事や手法適用の為には問題領域毎に部品整備が必要である事などが課題であった．これに対して本稿ではモデル充足ソフトウェア合成フレームワーク (Model Satisfiable Software Synthesis Framework, MSSS フレームワーク) を提案する．MSSS フレームワークは生成コードの信頼性を B Method の枠組みを応用して保証し，部品の整備を既存ソフトウェアからの部品自動生成により容易にする．MSSS フレームワークの全体像を図 3.1 に示す．MSSS フレームワーク は大きく分けて‘モデル充足ソフトウェア合成 (Model Satisfiable Software Synthesis, MSSS)’ と‘モデル充足細粒度部品生成 (Model Satisfiable Fine-graned Component Generation, MSFC 生成)’ からなる．また，MSSS は部品再利用に基づくコード生成手法であり図のようにモデル充足細粒度部品 (Model Satisfiable Fine-graned Component, MSFC) を部品リポジトリに蓄積する．

図 3.1 左側のように MSSS は要求を記述した B Method のモデルを入力として，部品を再利用することによって要求モデルを満たす B Method のソフトウェアを生成する．また，図 3.1 右側のように MSFC 生成は B Method のソフトウェアを入力として，そこから部品を自動生成してリポジトリに登録する．図 3.1 中央のように MSFC は B Method のモデルと実装にあたる実装依存モデルと細分化実装で構成され，部品の仕様として細分化モデルが与えられる．MSFC では B Method の枠組みを応用して部品が仕様を満たすことを静的に保証できる．

MSSS フレームワークは既存の自動コード生成に対して以下のような特徴を持つ．

抽象度の高い入力 MSSS は B Method のモデルを入力としてソフトウェアを生成する．モデルはシステムが守るべき制約と操作の入出力関係のみが記述された抽象度の高い記述である．そのため，既存のコード生成と異なり，アルゴリズムやアルゴリズ

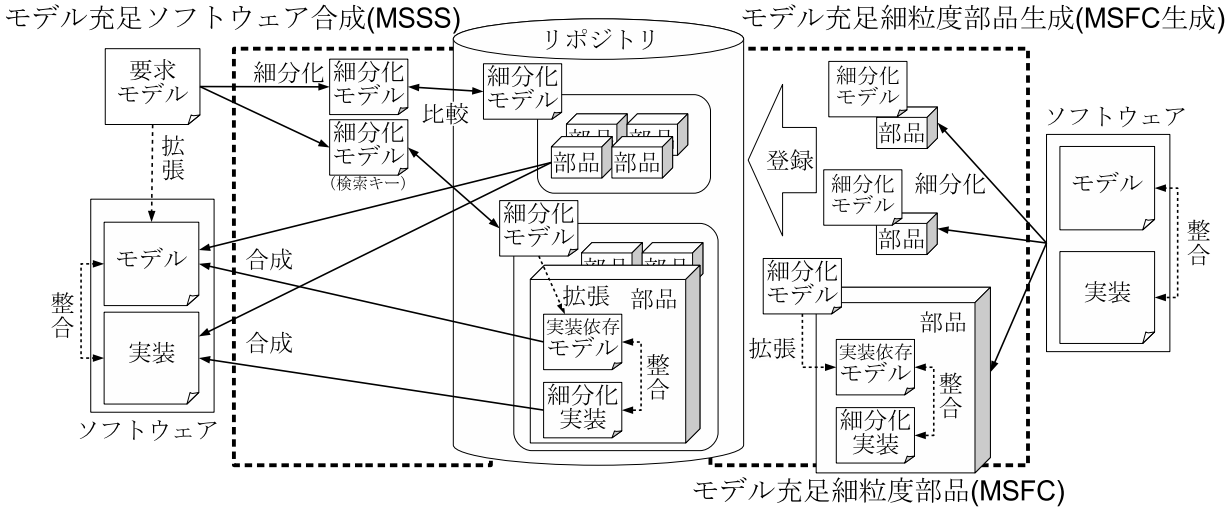


図 3.1: モデル充足ソフトウェア合成フレームワーク の全体像

ムに付けられた名前を記述せずにコードを生成できる．また，図 3.1 の細分化モデル間の比較を数学的判定で行うことで，変数や操作の名前に非依存な部品再利用ができる．これにより非妥当な命名や名前の解釈による誤りが生じない．

静的な信頼性保証 MSFC は B Method の枠組みを応用して部品が数学的仕様を満たすことを静的に保証できる．MSSS では MSFC に対して数学的仕様を検索キーとした健全性の高い部品再利用を適用することで生成されたソフトウェアの信頼性を静的に保証できる．

容易な部品整備 図 3.1 右側のように MSFC 生成により既存ソフトウェアから部品を自動生成することで部品リポジトリの整備を容易にする．部品に形式仕様を与える既存の手法では部品が仕様を満たす事を保証するのに大きなコストを要したが，MSFC 生成では生成される部品が仕様を満たすことを保証できる．

MSSS フレームワークは‘既存の自動コード生成の問題点を克服するためのアプローチ’であるとともに，‘形式手法において開発を自動化するアプローチ’ でもある．B Method をはじめとする形式手法は不具合が人命や多額の損失につながるクリティカルなシステム開発に利用されてきたが，形式手法の導入には大きなコストを必要とすることから一般のシステム開発への利用は十分とは言えない．これに対して MSSS フレームワークにより形式手法におけるソフトウェア開発を自動化することでより一般的なシステム開発に形式手法を適用することが期待できる．

3.2　モデル充足ソフトウェア合成フレームワーク の構成

本節ではモデル充足ソフトウェア合成フレームワーク を構成する**モデル充足細粒度部品 (MSFC)**，**モデル充足ソフトウェア合成 (MSSS)**，**モデル充足細粒度部品生成 (MSFC 生成)** の概要とそれぞれの関わりを説明する．なお，本節の説明は概要にとどめ，MSSS の詳細は 6 章で，MSFC の詳細は 4 章で MSFC 生成の詳細は 7 章で説明する．

MSSS は図 3.1 左側のように入力された B Method のモデルを部品と同じ粒度に細分化し，細分化された形式仕様を検索キーとした健全性の高い部品検索を行い，得られた部品群を細分化と逆の手順で合成することによりソフトウェアを合成する．要求仕様に対して部分一致する部品の仕様を検索するのではなく，この様に細分化された要求仕様と完全一致する部品を検索することは検索に要する計算量を削減する上で有利に働く．

MSSS は部品再利用でソフトウェアを合成するため，必要な部品が部品リポジトリに存在しない場合には部品を追加する必要がある．部品の追加には人手による部品の記述が必要であるが部品の追加を繰り返すことでいずれはリポジトリが充実して部品の追加なしにソフトウェアを生成できると考える．このためには追加された部品が他のソフトウェア生成時にも再利用できる必要があり，部品の再利用性向上が重要となる．2.4.1 で述べたように再利用性向上には部品の粒度を細かくする必要があり，本手法では MSFC の粒度を細粒度とする．

MSFC は図 3.1 のように実装依存モデルと細分化実装からなり，MSFC の仕様として細分化モデルが与えられる．MSFC には B Method を応用しており，細分化モデルと実装依存モデルは B Method のモデルにあたり，細分化実装は B Method の実装にあたる．B Method のモデルと細分化モデルの違いは粒度にある．大域変数を持つ B Method のモデルは必ず初期化を持つが，細分化モデルは常に初期化を持たず，また，操作で変化する大域変数も 1 変数のみに限定される．不変条件や事前条件などの制約条件についても細分化モデルは操作を表すのに必要最低限な制約のみを持つ．これにより部品の粒度を細粒度化する．

MSFC では細分化実装を細分化モデルに対する実装とせず，実装依存モデルに対する実装とする．これは B Method におけるモデルと実装間の整合性の定義が部品の仕様と実装間の間柄を表すには不都合があるためである．2.5.3 項の式 (2.6) に示した証明責務が真であるときに実装操作はモデルを満たすが，この証明責務において実装操作の代入文 V_{Ij} は輸入するモジュール操作の事前条件を含む．例えばファイルを利用して集合を保存するモジュールであれば，‘ファイルが開いてなければならない’ などの制約が事前条件として与えられる．このような証明責務を真にするために B Method のモデルは不変条件や事前条件に‘ファイルが開いている’ といった実装依存の表現を持つ．このため，B Method の枠組みで部品の整合性を保証するためには実装依存モデルのような実装依存の制約を持ったモデルが必要になる．MSFC では実装依存モデルと細分化実装の組を部

品とし，実装依存の制約を含まない細分化モデルを部品の仕様として与える．

MSFC 生成は図 3.1 右側のように既存のソフトウェアを細分化して MSFC とその仕様である細分化モデルを生成し，部品リポジトリに登録する．先に述べたように MSSS では要求モデルを部品と同じ粒度に細分化し，細分化された形式仕様を検索キーとして部品を検索する．このため，MSFC 生成で生成される細分化モデルと MSSS で要求モデルから得られる検索キーの粒度は等しくなければならない．本手法では MSSS で要求モデルから検索キーを生成するアルゴリズムと MSFC 生成で入力ソフトウェアのモデルから細分化モデルを生成するアルゴリズムを共通化することで MSSS の検索キーと等しい粒度の部品群を生成する．また，細分化モデルに合わせて入力されたソフトウェアを細分化して MSFC を生成するが，2.5.2 項で示したようにモデルの代入文が関数型言語と同様に代入順序が無いのに対して実装の代入文は命令型言語と同様に代入順序を持つ．この様に計算パラダイムが異なるモデルと実装の代入文から細分化モデルと同じ機能を持った実装の代入文を得るために，本手法では 2.4.2 項で紹介したプログラムスライシングを応用する．また，MSFC 生成で得られた実装依存モデルと細分化実装において証明責務が真になることを保証するために，制約条件の細分化ではソフトウェアを直接細分化せずに証明責務に変換した上で証明責務が真であることを保つよう証明責務を最小化し，最小化された証明責務から実装依存モデルと細分化実装の制約条件を生成する．

上述のように MSFC 生成 で得られた部品を MSSS で再利用するには部品の仕様であり，また，検索キーでもある細分化モデルを同一のアルゴリズムで生成する必要がある．本研究ではこの B Method のモデルからの細分化モデル生成アルゴリズムを**モデル細分化**と呼ぶ．細分化モデルは部品の仕様であるため，モデル細分化により矛盾などの誤りが生じると得られるソフトウェアの信頼性を保証できない．また，部品の再利用性は細分化モデルの粒度に依存するため，細粒度になるようモデル細分化を定める必要がある．また，膨大な部品群を効率的に検索するには検索キー，及び検索対象となる細分化モデルを検索に最適化する必要がある．この様に，モデル細分化は MSSS と MSFC 生成の両方で用いられ本研究の中核をなす技術である．

本稿では MSSS フレームワークを MSFC(4 章)，モデル細分化 (5 章)，MSSS(6 章)，MSFC 生成 (7 章) に分けて説明する．

3.3　運用上の制約

3.3.1　対象とする問題領域

MSSS フレームワークは B Method のモデルを入力としてソフトウェアを生成するため，MSSS の入力となる要求は B Method のモデルで記述できるものに限定される．また，信頼性保証の枠組みとして B Method を利用するため MSSS で保証される信頼性は

B Method の枠組みで保証できる信頼性に限定される．そのため，以下のように問題領域が限定される．

1. 要求するシステムの状態は集合論と述語論理で記述できる．
2. 要求する操作は戻値を返すかシステムの状態を変化させる．
3. 操作の開始から完了までの処理時間を要求や制約として与えない．
4. 1 操作の途中で他の操作により大域変数が変化する様な非同期処理は生じない．また，開始した操作は割り込みを受けること無く完了する．
5. トランザクションの開始と終了やファイルの開閉に挟まれた 1 連の操作は要求として与えない．

制限 (1) から制限 (4) は B Method のモデルに課せられた制限であり，MSSS がモデルを入力とする事に起因する．制限 (1) と制限 (2) は出納や物品管理システムの開発などを問題領域として想定するが, ユーザインタフェースや処理時間に対する制約や非同期処理や割り込みは想定しない事を意味する．また，制限 (3) により実時間制御システムへの手法適用は困難である．ただし，B Method の適用事例として衝突などの危険な状態が生じないことを保証した列車運行システムがあり, 処理時間を考慮しなければ制御システムにも応用できる. 制限 (5) は完結したトランザクションは操作の開始と終了時に値が変化しないため，実行順序の存在しない要求モデルには要求として記述できないことに起因する．ただし，1 変数が複数回変化する 1 連の操作を複数の操作に分割する事でトランザクションを含むソフトウェアを生成できる. 例えば, トランザクションの開始と終了をそれぞれ独立した操作として, トランザクションの開始と終了の間でのみ実行される事を事前条件で規定することで制限 (5) は守られる.

3.3.2　記法と運用の制限

本研究では MSSS フレームワークで扱う実装を以下のように制限する.

1. リンク不変条件は ‘抽象変数 =(実装変数の式)’ の形式に限定される.
2. MSFC で利用するモジュールは単一の概念に対する操作を集約する．また，モジュールが提供する各操作は提案手法でそれ以上細分化できない.
3. WHILE ループの中で大域変数への代入を行ってはならない.
4. 実装の操作において大域変数に代入した後にその大域変数を参照してはならない.
5. 実装の操作において大域変数への代入時に参照される式は局所変数として格納できるものに限定する. ただし, 代入される大域変数自身を参照することは可とする.

文法制限 (1) はモデルと実装の大域変数の対応付けを容易にするためである. これによりモデル変数と実装変数の対応が多対多である実装を書けないが, 詳細化を十分に行ったリ

ファインメントを入力モデルとすることで 1 対多, あるいは 1 対 1 で記述できると考えられる.

文法制限 (2) は部品選択 (6.3 節) で作業者によるモジュール選択を容易にするためである. 単一の概念に対する操作とは ‘ある機械 A を制御する操作’ や ‘集合をファイルで扱う操作’ などであり，これによりモジュールが扱う問題領域や操作対象の把握を容易にする. また，モジュールの各操作は MSFC の粒度を超えないものとする．これは，モジュールの各操作の粒度が大きいと細分化実装を細分化モデルの粒度に収めることができず，不整合を生じるためである．なお，MSFC では条件節と代入を分離しないため，条件節を排除することでモジュールの操作の粒度は MSFC より小さくなりうる．本稿で想定するモジュールは ‘入出力’，‘永続化’，‘制御’ に分類される．入出力では標準入出力，パイプ，各ネットワークプロトコルの実装に対する集合と写像の入出力モジュールなどが考えられる．また，永続化ではファイル，データベースに対する集合と写像の永続化モジュールなどが考えられる．また，制御では問題領域で制御対象となる機械ごとにモジュールを用意する．例えば，エレベータであれば，ボタンのランプ制御，ドアの開閉制御，モータ制御などのモジュールが考えられる.

文法制限 (3), (4), (5) は部品を合成した際の副作用を解消するためである. (3), (4) の制限は局所変数を活用することで解消出来ると考えられる. また, (5) の制限はモジュールにおけるデータ表現を制限するが代替モジュールを用意することで解消できる. また, B Method ではモデルに対して段階的詳細化を行うため, モデルと実装の他にリファインメントが登場するが, 本稿の提案手法ではリファインメントの書式を入出力として扱わない. そのため, 本研究で部品自動生成への入力となるソフトウェアのモデルは十分に詳細化が行われたリファインメントをモデルとして書き下したものを想定している. モデルとリファインメントの間の証明責務はモデルと実装の間のそれとほぼ同じであるため, 本研究を応用してリファインメントを細分化し部品に含めることが考えられる.

3.3.3　想定する運用

MSSS フレームワークは部品再利用に基づくソフトウェア自動生成手法であるため，部品リポジトリに必要な部品が存在しないと利用者が部品の追加をしなければならない．そのため，MSSS フレームワークを運用する上で部品リポジトリの整備が重要である．MSSS フレームワークでは図 3.2 のような運用により部品リポジトリを整備する．部品リポジトリの立ち上げには MSFC 生成を用いる．部品リポジトリの立ち上げはリポジトリが存在しない問題領域 A に対して問題領域 A における B Method のソフトウェア群を入力として MSFC 生成により部品群を生成し，空のリポジトリに部品群を登録する．この様にして立ち上げた部品リポジトリにおいて MSSS により問題領域 A のソフトウェアを合成する．不足部品がある場合は成果物であるソフトウェアの実装を直接記述せずにソフトウェ

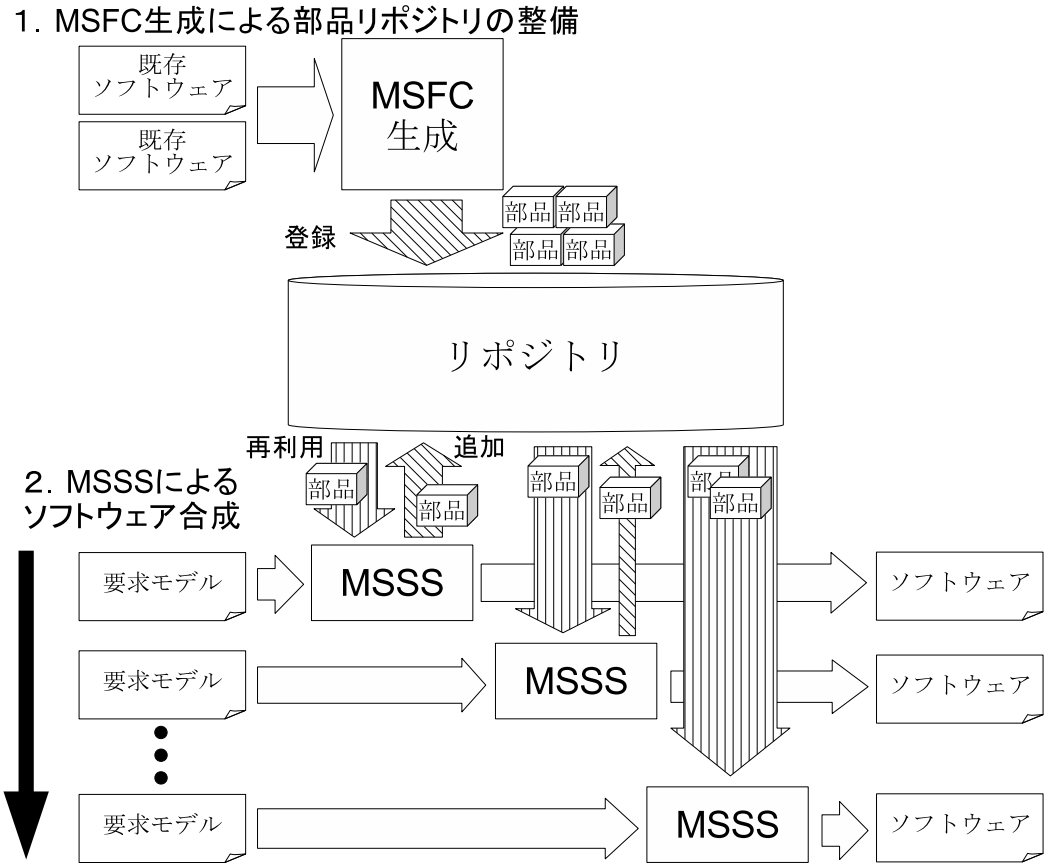


図 3.2: MSSS フレームワークの想定する運用

アを合成するのに足りない部品を記述する．このようにして記述された不足部品は全て部品リポジトリに登録することでソフトウェア開発を繰り返す毎に部品リポジトリが充実するため図のように不足部品が減少し，最終的には部品の追加なしにソフトウェアが合成できると期待できる．同一の問題領域で MSSS が繰り返される要因としては一度生成したソフトウェアの機能拡張や仕様変更の他にも，顧客毎に要求に応じたカスタマイズが必要な業務システムの開発などが挙げられる．

この運用は MSSS によるソフトウェア開発の前段階としてリポジトリの整備を行い，問題領域に共通する部品群を整備する点でソフトウェアプロダクトライン (SPL) に近い．しかし，共通部品の整備を MSFC 生成により自動化することで既存の SPL に比べてプロジェクトの立ち上げに要する作業量と時間を圧縮でき，また，整備された部品の信頼性も保証できる．

SPL では部品を再利用するが，各部品を連動させるためのコードや部品として存在しないコードは開発者が記述する．このため，部品を連動させるコードや部品化しても再利用頻度が不十分であると判断されたコードは複数のソフトウェアで繰り返し利用者が記述することになる．これに対して提案手法では利用者が合成された記述を直接修正することは一切無く，コーディングは足りない部品の記述とリポジトリへの追加によって行われる．このため，SPL に比べてコードの再利用性が高く，一度記述した実装は 2 度記述することは無い．この様に部品の追加を繰り返すことにより部品リポジトリが成長し，それに伴って不足部品は減少し，最終的には部品の追加なしでソフトウェアが合成できると期待できる．

既存の部品を自動で再利用して新たなソフトウェアを構築するという点で，MSSS は Web サービスの合成 と共通の目的を持つ．Web サービスの合成 では形式的に記述された要求仕様を満たすよう，複数の Web サービスを自動で結合して新たなサービスを提供する [21, 19]．しかし，要求仕様に記述する型に注目すると MSSS と Web サービスの合成はその目的が異なる．MSSS の要求モデルではデータ型は抽象データ型で表される．これは，データ間の関係や，適用される演算に注目したデータ型であり，実世界の概念との関連付けを明示しない．一方で Web サービスの合成ではデータ型が実世界の概念を反映しており，たとえ同じ「貸し出し機能」を提供するサービスでも再利用の可否は貸し出す対象が「車」か「本」かなどに依存する．この様に，‘与えられた値段以下の車のリスト’ を返すサービスを要求する際には，サービスが車の価格を持つことが重要である．一方で本研究で対象とするソフトウェア合成では ‘集合 A から整数への関数において，値域が引数以下の部分関数’ を返す機能が重要であり，車であることは重要でない．

第4章

モデル充足細粒度部品

4.1 概要

本章では図3.1のMSSSフレームワークで用いられるモデル充足細粒度部品(MSFC)を定義する．2.4.3項で述べたようにソフトウェア部品再利用では部品に仕様を付加することが重要である．B Methodではモデルがソフトウェアの仕様にあたるため，B Methodの実装をソフトウェア部品，B Methodのモデルを部品の仕様とすることが考えられる．しかし，2.5.2項で述べたようにモデルと実装間の整合性を保証するにはモデルに実装依存の表現を与えねばならず，仕様の抽象度が低下する．そのため，本研究ではB Methodのモデルと実装にあたる**実装依存モデル**と**細分化実装**でソフトウェア部品を構成し，さらに，実装依存モデルから実装依存の表現を取り除いた**細分化モデル**を部品の仕様とする．この細分化モデルの粒度を細粒度にした部品を**モデル充足細粒度部品 (Model Satisfiable Fine-graned Component, MSFC)** と定義する．MSFCでは実装依存モデル M_D と細分化実装 M_I において $M_D \sqsubseteq M_I$ であり，さらに実装依存モデル M_D が細分化モデル M_A に対して4.2.2項で定義する**拡張**である．この細分化モデルと部品の関係を**充足**と定義し，部品が細分化モデルを充足すると言う．

4.2 記述の定義

4.2.1 細分化モデル

細分化モデルは式(4.1)のようにパラメタ制約 C_A, プロパティ制約 P_A, 不変条件 I_A, 事前条件 Q_A, 代入 V_A で構成され，図4.1の様にB Methodのモデルに似た記述を持つ．

$$(C_A, P_A, I_A, Q_A, V_A) \tag{4.1}$$

式(4.1)が細分化モデルであるとは代入 V_A が細粒度な代入であり, 制約条件 C_A, P_A, I_A, Q_A が V_A を表すのに不要な制約を持たない事を言う．詳細は5章で説明するが，‘V_A を

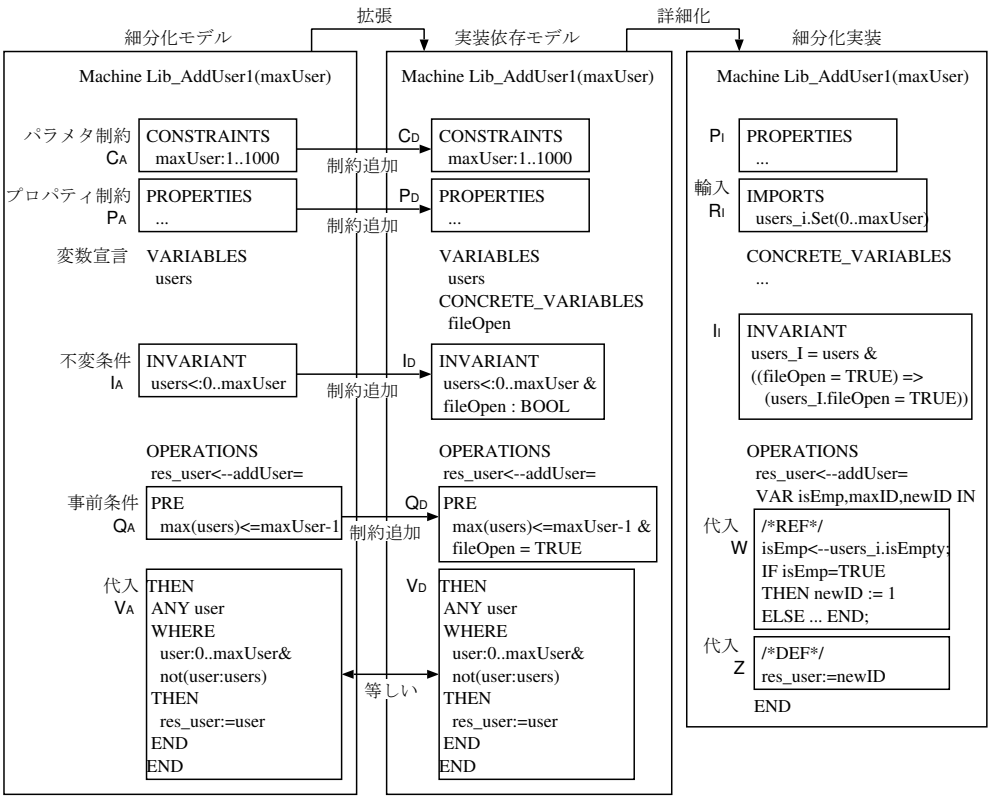


図 4.1: MSFC の記述例

表すのに不要な制約' とは V_A に現れる変数以外を含む制約である．また，同じく詳細は 5 章で説明するが，細粒度な代入 V_A を本研究では以下のように定義する．

1. 1 変数のみ変化する．
2. ANY 文は非決定的値の戻値への代入のみを持つ．
3. SELECT 文の条件群は互いに非決定的である．

式 (4.1) の様に制約条件と代入の組を細分化モデルとすることで，細分化モデルで代入の機能を表現できる．代入の機能を表現するのに代入 V_A だけでなく，制約条件が必要なのはモデルにおいて代入 V_A がどの様に機能するかは代入単体では決定せず，制約条件で代入を構成する変数間関係を定める必要がある為である．また，初期化は事前条件が常に真である操作と捉える事ができ，初期化に関する細分化モデルは $(C_A,\ P_A,\ I_A,\ \mathrm{true},\ U_A)$ と表される．式 (4.1) は 2.5.3 項のモデルの証明責務に必要な要素を全て持つため無矛盾性を保証できる．

4.2.2　実装依存モデル

実装依存モデルは式 (4.2) のようにパラメタ制約 C_D, プロパティ制約 P_D, 不変条件 I_D, 事前条件 Q_D, 代入 V_D で構成され，図 4.1 の様に細分化モデルに似た記述を持つ．

$$(C_D, P_D, I_D, Q_D, V_D) \tag{4.2}$$

MSFC において細分化モデルと実装依存モデルは式 (4.3) の関係を持つ．これは細分化モデルに実装依存の制約を論理積で追加した記述が実装依存モデルであることを意味する．すなわち, 実装依存モデルは細分化モデルと同じ機能も持つが, より実装に特化した記述である．このことを, 本稿では実装依存モデルは細分化モデルに対する**拡張**であるとする．また，これにともない実装依存モデルに追加された変数を**実装依存変数**と呼ぶ．実装依存変数はファイル開閉のフラグなどであり，細分化実装のモジュールにより生じる．

$$(C_D \Rightarrow C_A) \wedge (P_D \Rightarrow P_A) \wedge (I_D \Rightarrow I_A) \wedge (Q_D \Rightarrow Q_A) \wedge (V_D = V_A) \tag{4.3}$$

4.2.3　細分化実装

細分化実装は式 (4.4) のようにプロパティ制約 P_I, 輸入 R_I, 不変条件 I_I, 代入 W, Z の組で構成され，図 4.1 の様に B Method の実装に似た記述を持つ．

$$(P_I, R_I, I_I, W, Z) \tag{4.4}$$

代入 W は'参照部' といい大域変数を参照し計算結果を局所変数に格納する代入である．また, 代入 Z は'代入部' といい参照部 W で計算された局所変数の値を大域変数に格納す

る代入である．ここで, 参照部 W では大域変数を変更してはならず, 代入部 Z では大域変数を参照してはならない．この様に代入文を参照部と代入部に分割するのは実装の操作をモデルの操作と同様に結合するためである．例えば, 細分化モデルの操作が V_{A_1}，細分化実装の参照部と代入部が W_1, Z_1 である部品 1 と V_{A_2}, W_2, Z_2 である部品 2 に対して, 細分化モデルの操作を同時代入で結合した操作 $(V_{A1} \parallel V_{A2})$ に対応する実装操作を得る事を考える．部品 1 の代入に部品 2 の代入を単純に結合すると $(W_1;\ Z_1;\ W_2;\ Z_2)$ が得られるが, 部品 1 の代入部 Z_1 で変更される大域変数を部品 2 の参照部 W_2 が参照する場合, この計算結果は $(V_{A1} \parallel V_{A2})$ と異なる．本稿ではこの問題を ‘副作用’ と呼び, $(W_1;\ W_2;\ Z_1;\ Z_2)$ のように代入部の前に参照部を置くことで副作用を解消する．図 4.1 の細分化実装では`/*REF*/`が参照部, `/*DEF*/`以降が代入部である．

4.3　信頼性の定義

MSFC では B Method に習い, 証明責務を用いて信頼性を定義する．細分化モデル, 実装依存モデル, 細分化実装がそれぞれ式 (4.1), 式 (4.2), 式 (4.4) である MSFC の証明責務を式 (4.5), 式 (4.6), 式 (4.7) に示す．これらはそれぞれ初期化を行う細分化モデルの無矛盾性, 操作を行う細分化モデルの無矛盾性, 実装依存モデルと細分化実装の整合性を表す．これらを証明することで部品仕様の無矛盾性と部品の整合性を保証する．さらに実装依存モデルは細分化モデルに対する拡張であるため, 証明責務が真である高信頼ソフトウェア部品は無矛盾な仕様を満たす．

$$C_A \wedge P_A \Rightarrow [V_A] I_A \tag{4.5}$$

$$C_A \wedge P_A \wedge Q_A \wedge I_A \Rightarrow [V_A] I_A \tag{4.6}$$

$$(C_D \wedge P_D \wedge I_D \wedge Q_D) \wedge (P_I \wedge I_I) \Rightarrow [W;\ Z] \neg [V_A] \neg I_I \tag{4.7}$$

4.4　部品リポジトリ

4.4.1　概要

本節では MSFC を保存する部品リポジトリについて説明する．本研究では MSFC 生成により部品リポジトリに格納される部品数が膨大になるため，再利用が容易なリポジトリ構成が必要である．本研究では部品再利用を容易にするために ‘検索が容易なリポジトリの構造’ と ‘部品の結合を容易にする情報の付加’ について説明する．

部品検索はリポジトリに格納された部品と検索キーの間でマッチングを行うことで検索キーにマッチする部品を得る．しかし，これを単純に行うと部品数だけマッチングを

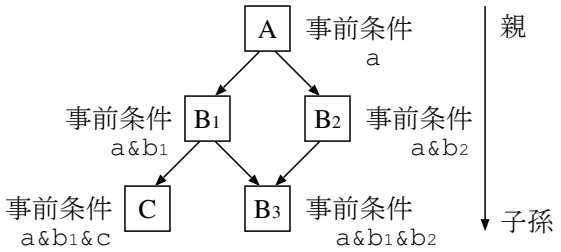


図 4.2: リポジトリにおける細分化モデルの階層構造

行うことになり，膨大な部品を格納したリポジトリにおいて検索が困難になる．このような部品数に対するスケーラビリティを向上するために，本研究では部品間に階層構造を与えて部品数に対する判定回数の増加を抑制する．このリポジトリの構造については 4.4.2 項で説明する．

部品の結合の詳細は 6 章で説明するが，本稿で提案する MSSS では各部品の変数名を統一することで各部品が連携して動作するようにする．この変数名の統一は部品の変数名を要求モデルの変数名に置換することで行うが，実装の変数名を統一するためには部品において実装依存モデルと細分化実装間の変数の対応が必要になる．本手法ではこの変数の対応をモデル実装間変数対応としてリポジトリに格納する．モデル実装間変数対応については 4.4.3 項で説明する．

4.4.2　リポジトリの構造

MSFC は細分化モデルをインデックスとしてリポジトリに格納される．このインデックスに対して細分化モデルを検索キーとしたマッチングを行うことで部品を検索する．この部品検索については 6.2 節で詳しく説明するが，ここでは細分化モデル M_C が細分化モデル M_K を検索キーとして得られる事を $M_C \subseteq_S M_K$ と表す．このマッチングを用いて部品リポジトリの階層構造を‘$M_1 \subseteq_S M_2$ ならば M_2 は M_1 の子孫である’と定義する．例えば，$B_1 \subseteq_S C$, $B_1 \subseteq_S B_3$, $B_2 \subseteq_S B_3$, $A \subseteq_S B_1$, $A \subseteq_S B_2$ という関係をもつ細分化モデル A, B_1, B_2, C は図 4.2 のような階層をなす．

本研究におけるマッチングは $M_1 \subseteq_S M_2 \wedge M_2 \subseteq_S M_3 \Rightarrow M_1 \subseteq_S M_3$ という推移律を持つ．また，$M \nsubseteq_S K$ ならば $M \subseteq_S X$ である任意の X について $X \nsubseteq_S K$ である．このため，上記のように階層構造を与えることで検索の分枝限定が可能になる．例えば，検索キー K に対して $A \nsubseteq_S K$ であるならば，細分化モデル A の子孫に $X \subseteq_S K$ である細分化モデル X は存在しない．同様に図 4.2 において $B_2 \nsubseteq_S K$ ならば，B_3 は検索キー K に対してマッチしない．

上記の判定が可能なのは，‘$M \nsubseteq_S K$ ならば $M \subseteq_S X$ である任意の X について $X \nsubseteq_S K$ である’という規則による．この規則は‘検索キー K にマッチする細分化モデルの事前条

件が K の事前条件から任意の制約を取り除いたものである' という事前条件に関する部分一致でマッチングが行われる事による．例えば，事前条件として命題 a, b_2 を持つ細分化モデル B_2 の子孫は必ず事前条件として命題 a, b_2 を持つ．そのため，事前条件に命題 b_2 を持たない検索キー K に対して B_2 の子孫は何れもマッチしない．結果として，細分化モデル K を検索キーとした部品検索では細分化モデル A, B_2 をインデックスとして持つ部品群が得られる．

4.4.3　モデル実装間変数対応

MSSS で部品を再利用するために部品を登録する際に 'モデル実装間変数対応付け' をリポジトリに格納する．モデル実装間変数対応付けはモデル変数 x_A とその変数に対応する実装変数 x_I の組 (x_A, x_I) のリストであり各部品毎に保存される．モデル実装間変数対応付けは部品間で変数名を統一するのに用いる．部品結合時には各部品を連動させるために同じ情報を格納する変数の名前を部品間で統一する．これは部品の細分化モデルの変数名を要求モデルに合わせて書き換え, モデル実装間変数対応付けから細分化モデルの変数に対応する実装変数を特定して変数名を書き換える事で行う．部品検索は要求モデルから得た細分化モデルと部品の細分化モデルのマッチングであるため, 検索で得た部品の細分化モデルと実装依存モデルの変数名を要求モデルの変数名に書き換えることは容易である．モデル実装間変数対応は MSFC 生成の部品登録時に対応付けを行い，部品と共にリポジトリに格納する．この対応付け手順は 7.3.3 項で説明する．

第5章

モデル細分化

モデル細分化はB Method のモデルを細分化して細分化モデル群を得る．モデル細分化はMSSS では要求モデルから検索キーを生成するために，MSFC 生成では入力モデルから部品の仕様を生成するために利用される．MSFC 生成で得られた部品群がMSSS で再利用できるのは双方で同じモデル細分化を用いており，検索キーと部品の粒度が等しいことが保証されているためである．この事からモデル細分化はMSSS フレームワークの中核技術と言える．本章ではモデル細分化の概要，および詳細手順を説明し，得られる細分化モデルの信頼性を定理証明に基づいて評価する．

5.1 概要

モデル細分化はB Method のモデルを細分化して細分化モデル群を得る．B Method のモデルは1つの初期化 U と複数の操作 (Q_k, V_k) を持つ．ここでは解説を簡単化するために1つの操作にのみ注目し，モデルの制約と操作の組 (C, P, I, Q, V) を細分化することを考える．初期化 U は事前条件が存在しない，すなわち，Q が TRUE であるとして一般化する．

モデル細分化は (C, P, I, Q, V) を細分化して細分化モデル群 $(C_i, P_i, I_i, Q_i, V_i)$ を得る．V が初期化であるならば細分化モデルは式 (4.5) の初期化の整合性を満たし，V が操作であるならば細分化モデルは式 (4.6) の操作の整合性を満たす事が期待される．細分化モデルの代入 V_i は入力モデルの代入 V を細粒度な代入に分割したものである．代入が細粒度であるとは4.2.1 項で定義したように，1変数のみを変化させ，ANY 文は非決定的値の戻値への代入のみを持ち，SELECT 文の条件群が互いに非決定的である事をいう．また，C_i, P_i, I_i, Q_i は V_i で用いられる変数間の関係を定めた制約である．

モデル細分化は図5.1 に示すように‘非決定的値生成操作の分離’，‘制約条件展開’，‘操作分割’，‘制約条件抽出’，‘構文要素整列’によって行われる．さらに，生成された細分化モデルに対して検証を行い，必要ならば修正をする．

以下に各手順の役割を説明する．

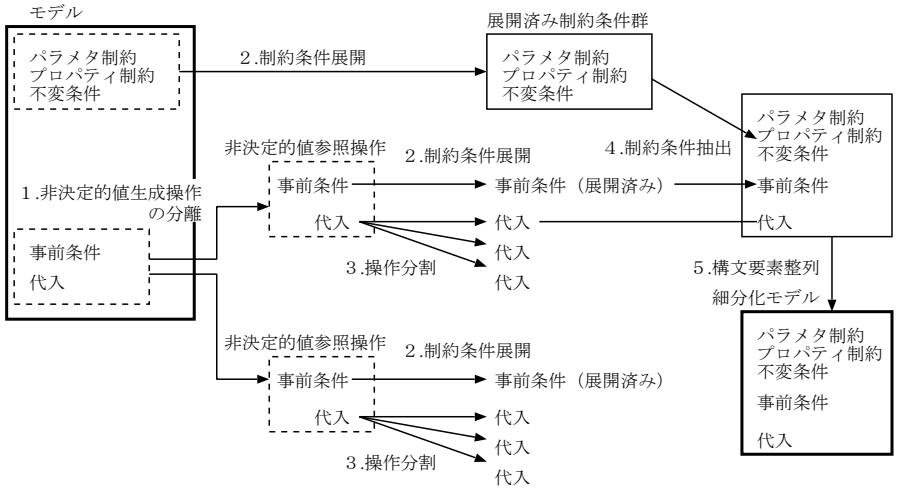


図 5.1: モデル細分化手順

1. **非決定的値生成の分離:** 事前条件と代入の組 (Q, V) から非決定的値生成操作 (Q_{AnyDec}, V_{AnyDec}) と非決定的値参照操作 (Q_{AnyRef}, V_{AnyRef}) を得る．非決定的値生成操作は V が含む ANY 文で局所変数に格納される非決定的な値を生成してそれを戻値として返すだけの操作である．非決定的値参照操作は V で ANY 文により生成される値を操作の引数として受け取り, V が含む値の生成以外の代入を行う操作である．詳細は 5.2 節で説明する．
2. **制約条件展開:** 制約条件 C, P, I, Q に対して推論を行い, 入力モデルでは明示されない変数間の関係を明示した制約条件 C_{rsn}, P_{rsn}, I_{rsn}, Q_{rsn} を得る．制約条件展開は‘等価な変数の特定’ と ‘制約条件抽出’ において与えられた変数間関係を特定するのに必要となる．詳細は 5.3 節で説明する．
3. **等価な変数の特定:** $x = dom(r)$ における x, r や $Z = X \cup Y$ における Z, X, Y の様に等号 $(=)$ で関係を定義された変数群を ‘等価な変数’ と定義し, 等価な変数の集合群 $E_i = \{e_{ij} \mid 1 \leqslant j \leqslant a\}$ を得る．ここで, e_{ij} は変数であり, 任意の x, y について e_{ix} と e_{iy} は等価な変数である．制約条件展開によって任意の 2 変数間の関係が展開されているため, ‘=’ で関係づけられた式を見つけることは容易である．
4. **操作分割:** 操作の代入 V と等価な変数群 $E_i = \{e_{i1}, \ldots, e_{ia}\}$ を入力として細分化された代入 V_i を出力する．ここで, V は非決定的値生成の分離で得られた非決定的値生成操作, あるいは非決定的値参照操作であり, V_i は等価な変数群 E_i に対する代入文を条件分岐の条件毎に抽出した代入文である．詳細は 5.4 節で説明する．
5. **制約条件抽出:** 展開された制約条件 C_{rsn}, P_{rsn}, I_{rsn}, Q_{rsn} と操作分割で得られた代入 V_i を入力として細分化モデルの制約条件 C_i, P_i, I_i, Q_i を得る．ここで, C_i, P_i, I_i, Q_i は C_{rsn}, P_{rsn}, I_{rsn}, Q_{rsn} が含む V_i で用いられる変数間の制約である．詳細は 5.5 節で説明する．

6. **構文要素整列:** 得られた C_i, P_i, I_i, Q_i および V_i の構文要素を整列することで，文字列一致による同値判定を可能にする．詳細は 5.6 節で説明する．
7. **検証と修正:** 整列済の C_i, P_i, I_i, Q_i および V_i から細分化モデルを生成する．V_i が初期化から得られた代入であるならば証明責務は常に真である．V_i が操作から得られた代入であるならば証明責務が偽になる場合があるため, 式 (4.6) の証明責務を定理証明器等の補助を受けて利用者が証明する. 証明責務が偽であるならば手順 (2) の推論において証明に必要な命題が推論に失敗しているため, それを制約条件に追加する.

3.2 節で述べたように MSSS フレームワークでは MSSS と MSFC 生成でモデル細分化アルゴリズムを共通化することで部品と検索キーの粒度を等しくする．このため，ここで提案するモデル細分化と同様の分割が実装に対しても可能でなければならない. その上で，3.2 節で述べたように MSSS フレームワークでは部品の粒度を極力細かくして部品の再利用性を向上させる．このとき，特に問題になるのがモデルにおける非決定的な条件選択である．SELECT 文などの条件選択は $P_1 \Rightarrow F_1 \mid\mid P_2 \Rightarrow F_2$ のように条件式 (例では P_1, P_2) と代入文 (例では F_1, F_2) で表現されるが，P_1 と P_2 が排他的でない場合には $P_1 \wedge P_2$ に対して F_1, F_2 のどちらの代入で実装しても良い．このため，$P_1 \Rightarrow F_1 \mid\mid P_2 \Rightarrow F_2$ というモデルの代入を単純に $P_1 \Rightarrow F_1$ と $P_2 \Rightarrow F_2$ に分割すると条件 $P_1 \wedge P_2$ においてモデルと実装間で齟齬が生じる．また, 4.2.1 項で述べたように MSFC は ‘1 部品では 1 変数のみ変化する’ を粒度の基準とするが，モデルにおいて同じ内容を保持する式は実装において 1 変数で表現される事が考えられるため, 等価な式を構成するモデル変数に対する操作は分割出来ない．これらの事から, 手順 (3) のように ‘等価な変数’ を特定し, 等価な変数毎に操作を分割する．この粒度に沿った操作分割は手順 (4) で行われる．本手法ではさらに細粒度化するために手順 (1) を定めた．ANY 文の値を生成する部分と値を用いて計算する部分は手順 (4) で分割することが出来ないため, 手順 (1) で分割することで細粒度化する．手順 (5) は操作で用いられた変数群に関する制約を抽出することで細分化モデルの制約を得る．以降の節では各手順の詳細を説明する．また，このモデル細分化手法の信頼性は 5.7 節で保証する．

5.2　非決定的値生成の分離

5.1 節に述べたように非決定的値生成の分離では事前条件と代入からなる操作 (Q, V) を入力として, 非決定的値生成操作 (Q_{AnyDec}, V_{AnyDec}) と非決定的値参照操作 (Q_{AnyRef}, V_{AnyRef}) を出力することで操作を細粒度化する．分割元の操作を図 5.2(a) に表す分割元操作は操作戻値 res , 操作引数 $args$, 事前条件 Q, 制約 W で規定される値群 $v1, \ldots, vk$ についての代入 V_{any} で構成される ANY 文，および ANY 文以外の代入文 V からなる．この時の非決定的値生成操作と非決定的値参照操作をそれぞれ図 5.2(b), 図 5.2(c) に示す．非決定的

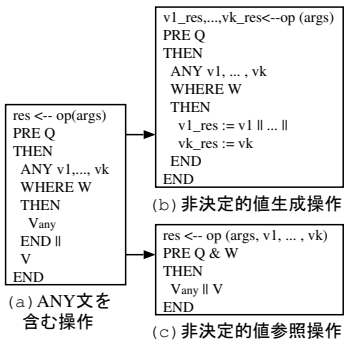


図 5.2: 非決定的値生成の分離

値生成操作は元の操作と同じ事前条件 Q を持ち, V_{any} の代りに局所変数 $v1, \ldots, vk$ を戻値にする代入を持った ANY 文を V_{AnyDec} とした操作である. この際, 局所変数毎に戻値となる変数を定義する必要があるが, 本稿ではこれを局所変数名の末尾に`_res` を加えたものとする. 非決定的値参照操作は事前条件 Q_{AnyRef} として元の事前条件 Q と ANY 文の制約 W の論理積 $Q \wedge W$ を持ち, 代入 V_{AnyRef} として ANY 文の代入 V_{any} と代入 V の同時代入 $V_{any} \mid\mid V$ を持つ. この際, 引数として ANY 文で定義された変数群を追加する.

上記の出力において非決定的値生成操作は ANY 文の制約で規定された非決定的な値を生成してそれを戻値とする操作であり, 元の操作が含む代入文は一切持たない. また, 非決定的値参照操作は事前条件で規定される非決定的値を引数として受け取り, 元の操作が含む代入を行う操作である. そのため, 非決定的値生成操作は ANY 文を持ち, 非決定的値参照操作は ANY 文を持たない操作となる. 非決定的値参照操作の事前条件に ANY 文の WHERE 部を加えることによって, 非決定的値生成操作の局所変数と非決定的値参照操作の引数を対応付けている. なお, ANY 文に類似する文法として充足代入文, 要素代入文, 局所定義文があるが, これらは全て ANY 文で表現できるため ANY 文に書き換えた上で分割する. 実装抽出では本節で行った分割と同様の分割を行う必要がある. しかし, ANY 文はモデルで用いることは出来るが実装で用いることが出来ないため, 7.2.3 項のように実装において非決定的値生成の分離を行う手順を与える.

5.3　制約条件展開

5.3.1　概要

制約条件展開は入力モデルに明示されない暗黙の制約条件を明示することで, 制約条件抽出で変数間の関係の取りこぼしを解消する. 例えば, 暗黙の制約条件とは $A \subseteq B \wedge B \subseteq C$ に対する $A \subseteq C$ などである. 制約条件抽出では細分化された操作が含む変数間の関係を抽出することで細分化モデルの制約条件を生成する. 上記の例では暗黙の制約条件 $A \subseteq C$

が明示されていないと細分化された操作が変数 A, C を含む場合に制約条件の取りこぼしが生じる．

制約条件展開はルールベースで推論を行う点で項書換えシステムと類似する．一般的な項書換えシステムの停止性を判定する一般的な方法は存在しないため [30]，単純な推論で制約条件を展開する場合，有限回で推論を打ち切る事で停止性を保証する必要がある．しかし，推論を打ち切りが生じると本来一致する命題数などに不一致が生じ，部品検索に失敗する．この問題に対して三鍋は B Method の演算子定義に着目し，制約条件を基本的な演算子のみで構成された命題に変換するプリミティブ化により，推論規則を推移律などに限定することを提案した [31]．本研究では制約条件展開にプリミティブ化を利用して推論規則を単純化することで，推論規則に対する網羅的な停止性検査を可能にする．以下に制約条件展開の手順を示す．

1. **プリミティブ化:** モデルに現れる演算子の種類を制限するために，与えられた式をより基本的な演算子 (**プリミティブな演算子**) のみで構成された式に書き換える．
2. **簡約化:** プリミティブ化により生じる集合定義などの入れ子構造を単純な式の論理積に展開する．
3. **推論による式の追記:** 推論規則の適用と式の追記を新たな式が推論されなくなるまで再帰的に適用する．
4. **主加法標準化:** 6.2.2 項のマッチングで部分一致を容易にするため，命題を主加法標準型から共通する命題をくくり出した形に統一する．

推論による式の追記は ‘書き換え’ ではなく式を ‘追記’ する点が項書換えシステムと異なる．一般的な項書換えシステムで式間の同値関係を判定するには停止性の他に ‘合流性’ を保証する必要がある [37]．合流性とは書き換え順序に依らず書き換え結果が一意に定まる性質であり，これが満たされない場合には等価な式から異なる字面が得られるため同値関係の判定に失敗する．これに対して推論による式の追加では全ての書き換え順序の試行結果を列挙すれば良いため合流性は必要なく，停止性のみを保証すれば良い．一方，プリミティブ化は追記ではなく書き換えを行うため，停止性と合流性の両方が必要である．

以下の項ではプリミティブ化，簡約化，推論による式の追記，主加法標準化について説明する．

5.3.2　プリミティブ化

プリミティブ化では項書換えによりモデル中の演算子の種類を減らすことで推論能力を損なうこと無く制約条件展開の推論規則を簡単化する．ここで，プリミティブ化後のモデルを構成する演算子をプリミティブな演算子と呼ぶ．本稿では文献 [1] にある B の

表 5.1: プリミティブ化の書き換え規則 (抜粋)

番号	書き換え元	書き換え後
1	$a \notin b$	$\neg(a \in b)$
2	$a \subseteq b$	$a \in \mathbb{P}(b)$
3	$a < b$	$a \leqslant b \wedge \neg(a = b)$
4	$a \nrightarrow b$	$\{r \mid r \in a \leftrightarrow b \wedge \mathrm{dom}(r) \subseteq a \wedge \mathrm{ran}(r) \subseteq b \wedge (r^{-1};\ r) \subseteq \mathrm{id}(b)\}$
5	$a \rightarrow b$	$\{r \mid r \in a \nrightarrow b \wedge \mathrm{dom}(r) = a\}$
6	$a \leftrightarrow b$	$\mathbb{P}(a \times b)$
7	$\mathrm{ran}(r)$	$\mathrm{dom}(r^{-1})$

演算子定義において他の演算子による定義を持たない演算子をプリミティブな演算子と定義する．また，プリミティブではない演算子をプリミティブな演算子に書き換える規則を B の演算子定義に基づいて整備する．ただし，B の演算子定義の一部は余りに素な要素に演算子を書き換えてしまい扱いにくいため，本稿では扱いやすい演算子の範囲内でプリミティブ化を行う．表 5.1 にプリミティブ化の書き換え規則の抜粋を示す．詳細なプリミティブ化の書き換え規則は A.1 に掲載する．

表 5.1 において書き換え 1 は否定的な演算子を $\neg$ を用いた表現に直す．表中では省略しているが，$\neq, \notin$ なども同様に書き換える．書き換え 2 は部分集合を巾集合の要素として書き換えている．書き換え 3 は不等号についての書き換えであり，$\geqslant, >$ や $\subset, \supseteq, \supset$ なども同様に $\leqslant, \subseteq, \neg(a = b)$ を用いた表現に書き換える．書き換え 4, 5, 6 は写像に関する書き換えであり，同様の書き換えを部分単射や全域単射などに対しても行う．なお，B の演算子定義では $\mathrm{dom}(r)$ をさらに集合的な表現に書き換えられるが量化子 $\exists$ などを用いた複雑な表記になり扱いが難しくなるため，本稿では dom をプリミティブな演算子としてそれ以上書き換えない．同様に B の演算子定義では整数値を集合を用いて定義しているが本稿ではこの書き換えを行わない．

プリミティブ化は B の構文要素定義に準じる事で合流性と停止性の保証を容易にしている．合流性が保証された代表的書き換え系として正則項書換え系がある [34] が，表 5.1 の書き換え規則は正則項書換え系の条件である ‘左線型である事’ と ‘重なりがない事’ を満たす．また，停止性の保証には書き換えがループしないことを保証する必要があるが，B の構文要素は定義にループが存在しないため停止性を保証できる．これにより構文要素の制限は合流性と停止性を持つ．

5.3.3　簡約化

プリミティブ化では表 5.1 の書き換え 4,5 の様に写像を集合の内包表現への書き換える．実際には書き換え 5 にはさらに書き換え 4 が適用可能なため，この集合の表現は入れ子が生じる．簡約化ではこの様な入れ子構造や命題の重複を解消する事で式の扱いを容易にする．この簡約化は $r \in \{f \mid P(f)\}$ を $P(r)$ に書き換えることで行う．また，冗長な式の簡単化のために $A \wedge A$, $A \vee A$ を A に，$(r^{-1})^{-1}$ を r に書き換える．

例えば $r \in a \to b$ という式はプリミティブ化により式 (5.1) の様に書き換えられる．この式に簡約化ルールを適用すると式 (5.2) が得られる．

$$\begin{aligned} r \in \{f \mid f \in \{g \mid g \in \mathbb{P}(a \times b) \wedge \mathrm{dom}(g) \in \mathbb{P}(a) \wedge \mathrm{dom}(g^{-1}) \\ \in \mathbb{P}(b) \wedge (g;\ g^{-1}) \in \mathrm{id}(b)\} \wedge \mathrm{dom}(f) = a\} \end{aligned} \tag{5.1}$$

$$r \in \mathbb{P}(a \times b) \wedge \mathrm{dom}(r) \in \mathbb{P}(a) \wedge \mathrm{dom}(r^{-1}) \in \mathbb{P}(b) \wedge \mathrm{dom}(r) = a \wedge (r;\ r^{-1}) \in \mathrm{id}(b) \tag{5.2}$$

5.3.4　主加法標準化

プリミティブ化では $P \wedge (Q \vee R)$ と $(P \wedge Q) \vee (P \wedge R)$ が異なる字面に変換される．これを項書換えで等価な字面に統一することは困難であるため，本稿ではシャノン展開を用いてこの字面を主加法標準型に統一する．シャノン展開は式 (5.3) のように $p_1 \ldots p_n$ で構成された論理式に対して $p_1 \ldots p_n$ とその否定の論理積を論理和で列挙した命題が得られる．

$$F(p_1, \ldots, p_n) = p_1 \wedge \cdots \wedge p_n \wedge F(t, \ldots, t) \vee \cdots \vee \overline{p_1} \wedge \cdots \wedge \overline{p_n} \wedge F(f, \ldots, f) \tag{5.3}$$

以下に論理和と論理積の展開手順を示す．

1. 制約条件において論理積と論理和を区切りとした命題のリストを $p_1, \ldots, p_n$ とする．
2. 制約条件に対して $p_1, \ldots, p_n$ を論理変数としたシャノン展開を適用する．式 (5.3) の $F(t, \ldots, t), \ldots, F(f, \ldots, f)$ が偽である命題は寄与しないため論理和から削除する．
3. $p_1, \ldots, p_n$ のうち，全ての論理和の項において常に肯定あるいは常に否定である命題群 P_I をくくり出し $P_I \wedge (Q_1 \vee \cdots \vee Q_k)$ の形にする．このくくり出しは次に行う推論による命題の展開の計算量を低減するために行う．推論は論理和毎に行うため，くくり出しにより推論の重複を回避できる．

5.3.5　推論による式の追記

推論による式の追記では構文要素の制限を適用した制約条件 P を入力として P において暗黙的な変数間関係を明示した命題 Q を出力する．以下に展開手順を示す．ここでは

表 5.2: 推論規則一覧 (抜粋)

番号	規則		
1	$X \in \mathbb{P}(Y) \land Y \in \mathbb{P}(Z)$	$\Rightarrow$	$X \in \mathbb{P}(Z)$
2	$(X \in \mathbb{P}(Y)),\ a \in X$	$\Rightarrow$	$a \in Y$
3	$(a = b),\ R(b, x)$	$\Rightarrow$	$R(a, x)$
4	($X \times Y$ が記述中に在る), $X \in \mathbb{P}(Z)$	$\Rightarrow$	$X \times Y \in \mathbb{P}(Z \times Y)$
5	$(S \in \mathbb{P}(Y),\ y \in Y),\ S = \emptyset$	$\Rightarrow$	$\neg(y \in S)$
6	$r[\{x\}] = \emptyset \land r[\{x\}] \in \mathbb{P}(\mathrm{dom}(r^{-1}))$	$\Leftrightarrow$	$\neg(x \in \mathrm{dom}(r))$
7	$(r \in \mathbb{P}(a \times b),\ (r;\ r^{-1}) \in \mathbb{P}(b)),\ y \in r[\{x\}]$	$\Leftrightarrow$	$y = r(x)$
8	$(r \in \mathbb{P}(a \times b),\ (r;\ r^{-1}) \in \mathbb{P}(b)),\ y \in r[\{x\}]$	$\Leftrightarrow$	$\{y\} = r[\{x\}]$
9	($\mathrm{id}(X)$ または $\mathrm{id}(Y)$ が記述中に在る), $X \in \mathbb{P}(Y)$	$\Rightarrow$	$\mathrm{id}(X) \in \mathbb{P}(\mathrm{id}(Y))$
10	$x \in \{a_1, a_2, \dots\}$	$\Leftrightarrow$	$x = a_1 \lor x = a_2 \lor \dots$

命題 P を論理積で分割して得られる命題の集合を S とする．

1. S に対して推論規則を適用して得られる命題群を T とする．推論規則の一部を表 5.2 に示す．表 5.2 は推論規則の抜粋である．
2. $S \cup T \neq S$ ならば $S \cup T$ を新たな S として 1 に戻る．
3. $S \cup T = S$ ならば $Q = \bigwedge S$ を出力する．

表 5.2 の推論規則において括弧で記述した命題はその規則を適用するための条件である．推論規則 $(G), P \Rightarrow Q$ を適用するとは，推論済の制約条件群が命題 G, P を含むならば命題 Q を推論することを言う．また，$(G), \neg Q \Rightarrow \neg P$ のように各推論規則の逆も適用する．例えば推論規則 2 の逆は $(X \in \mathbb{P}(Y)), \neg(a \in Y) \Rightarrow \neg(a \in X)$ である．5.3.2 項のプリミティブ化により $a \nrightarrow b$ の様な関数表現などが $a \times b$ のような積集合の表現に書き換えられる．これにより表 5.2 の推論規則では書き換えにより消失した演算子についての推論が不要となり，推論規則を簡略化できる．

表 5.2 の推論規則数は有限であるため，上記手順が停止しないならば推論規則間にループが存在する．ループの例としては表 5.2 の推論規則 6, 7, 8, 10 の様に '$\Leftrightarrow$' を持ち，推論元と推論結果が可逆な規則が挙げられるが，これらは同じ命題を繰り返し生成するため展開手順 (3) の終了条件によりループは停止する．そのため，本手法では新たな命題を生成し続けるループが存在しないことを示す事で停止性を保証する．

新たな命題を生成し続けるループの例として $x \Rightarrow x+0$ が挙げられる．この例では $x+0$ を新たな x として推論規則を適用することで，$x + 0 + 0 + \dots$ のように推論規則がループする．この様な無限ループは $x + 0$ の様に推論元の項より長い項を生成し，その式が他の推論規則中の 1 変数になりうる場合に生じ得る．本手法では推論規則 4 が X, Z から

推論される式 $X \times Y \in \mathbb{P}(Z \times Y)$ を $X \in \mathbb{P}(Z)$ と置くことで項を増やす無限ループとなる．しかし，推論規則 4 は‘$X \times Y$ が記述中にある’という条件を持つため，推論によって 2 項に増えた積集合を推論元として推論規則 4 を適用するには $A \times Z \times Y$ の様に 3 項以上の積集合が存在する必要がある．このため，推論規則 4 は無限ループせずに有限回で停止する．推論規則 9 についても同様に有限回で停止することを保証できる．

5.4　操作分割

操作分割では与えられた代入 V と注目する等価な変数群 $\{e_1, \ldots, e_a\}$ を入力として, 細分化された代入 $V'_1, \ldots, V'_l$ を出力する．$V'_1, \ldots, V'_l$ は変数群 $\{e_1, \ldots, e_a\}$ を変化させる代入文を条件分岐の条件毎に抽出した代入文である．このため, 1 つの代入 V から 1 つの等価な変数群 $\{e_1, \ldots, e_a\}$ について操作分割した結果は複数得られる．ただし, 2.5 節のモデルの代入文で説明したように条件選択 SELECT は非決定的な記述が可能であり, 排他的でない条件 P_1, P_2 について P_1 ならば代入文 S_1, P_2 ならば代入文 S_2 が与えられているとき, 実装においては必ずしも P_1 において代入文 S_1 が行われるとは限らない．そのため, 本手法では排他的でない条件選択については分割を行わない事とする．本手法では分割ルールを単純化するために, もっとも基本的な文法についてのみ分割ルールを用意し, 他の文法はこの基本的な文法に書き換える事で分割する．例えば, `x,y:=E1,E2` のような複数変数への等価代入は `x:=E1||y:=E2` のような同時代入と等価代入に書き換える．また, 有限非決定的選択文 (CHOICE), IF 条件文, 場合分け (CASE) は条件選択 (SELECT) に書き換える．以下に操作分割を関数 slice とした操作分割の再帰的定義を表す．ここで, 関数 slice は代入 V と注目する変数群 $\{e_1, \ldots, e_a\}$ を受け取り, 分割結果を代入文のリストとして返す．以下の slice を用いた表現では引数として注目する変数群 $\{e_1, \ldots, e_a\}$ を省略する．

等価代入 slice($x := E$) に対して, 変数 x が注目する変数群に含まれるならば, 結果は $x := E$ である．含まれないならば結果は空リストである．

同時代入 slice(V_1 || ... || V_n) に対して, slice(V_1), ..., slice(V_n) が条件選択を持たないならば結果は slice(V_1) || ... || slice(V_n) である．そうでないならば, slice(V_1), ..., slice(V_n) のうち排他的でない条件を持つもの同士を同時代入で結合した代入文のリストが結果である．ただし, 同じ SELECT 文から分割された SELECT 文は同時代入で結合せずに 1 つの SELECT 文にまとめる．これは排他的な条件で発火する代入文はそれぞれ別な部品になる事を意味する．

非決定的選択 (ANY) slice(ANY$v_1, \ldots, v_l$ WHERE W THEN V) を考える．W を論理積で分割して同じ変数を含む命題同士を論理積で結合した命題を $W_1, \ldots, W_k$ とする．$1 \leqslant i \leqslant k$ について V から W_i が含む局所変数群 X_i を含む代入文を抽出し同

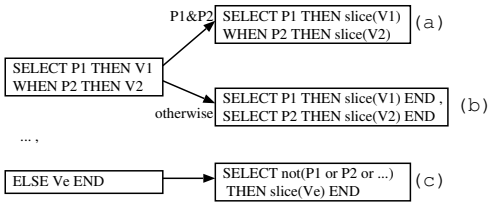


図 5.3: SELECT 文の分割

時代入で結合したものを V_i とする．また, $V_1, \ldots, V_k$ のうち注目する変数群に含まれる変数を変更する代入を $V_{j_1}, \ldots, V_{j_s}$ とする．この時, 出力は ANY $X_{j_1}, \ldots, X_{j_s}$ WHERE $W_{j_1} \wedge \cdots \wedge W_{j_s}$ THEN $(V_{j_1} \parallel \ldots \parallel V_{j_s})$ である．

条件選択 slice(SELECT P_{11} THEN V_{11} ... WHERE P_{1a} THEN V_{1a} ... WHERE P_{s1} THEN V_{s1} ... WHERE P_{sb} THEN V_{sb}) を考える．ここで, 任意の i に対して $P_{i1}, P_{i2}, \ldots$ は互いに排他的でない条件であり, $i \neq j$ である j に対して P_{ix}, P_{jy} は互いに排他的である．条件 P_{ix}, P_{iy} が排他的でないとは事前条件 Q に対して $Q \Rightarrow P_{ix} \wedge P_{iy}$ が真である事をいう．$SELECT_i$ を SELECT P_{i1} THEN slice(V_{i1}) WHERE P_{i2} THEN slice(V_{i2}) ... としたとき, 条件選択の分割結果は $SELECT_1, \ldots, SELECT_s$ である．ただし, slice(V_{ix}) が空であるならば P_{ix} も SELECT 文から取り除き, 全ての遷移条件が取り除かれるならば $SELECT_i$ は出力に含まれない．

B Method のモデルでは条件の評価も同時に行われるため, 互いに排他的でない遷移条件 P_1, P_2 とその代入 V_1, V_2 を持つモデルは P_1, P_2 どちらにも属する状態に対して V_1, V_2 のどちらの代入を行っても良いことを表す．すなわち, この様な排他的でない遷移条件を持つ場合には遷移条件 P_1 に対する実装が V_1 であるとは限らない．そのため, 本手法では排他的でない遷移条件は分割しない．図 5.3 に示すように, 条件 P_1, P_2 が互いに排他的で無いときは条件 P_1, P_2 を分割する事は出来ず (a) の様に一つの SELECT 文になる．条件 P_1, P_2 が互いに排他的であるときは (b) の様に条件 P_1, P_2 についての SELECT 文がそれぞれ作られる．ELSE 節については全ての条件と排他的であるため, 図 5.3(c) の様に全ての条件の論理和の否定を条件として SELECT 文を作る．

5.5　制約条件抽出

制約条件抽出は命題 G と 5.4 節の分割で得られた代入 V', および分割元の代入 V を入力として命題 G' を出力する．G は 5.3 節の制約条件展開で展開された命題群 C_{rsn}, P_{rsn}, I_{rsn}, Q_{rsn} であり, 命題 G' は V' で用いられる変数間の関係を定義する証明責務が真になる命題である．ここで, 関数 $vars(V')$ を代入 V' に含まれる変数群, 関数 $chs(G, X)$ を変

数群 X と任意のモデルパラメタ，およびプリミティブな値や型のみで構成される G に含まれる命題とする．また, V' で変化する変数群 X, V で変化するが V' で変化しない変数群 X', とする．この時, G が P_{rsn}, Q_{rsn} であるとき, $G' = chs(G, vars(V'))$ である．また, G が I_{rsn} であるとき, $G' = chs(G, vars(V') - X) \wedge chs(G, vars(V') - X')$ である．また，C_{rsn} については V', P', Q', I' に含まれるモデルパラメタと変数，およびプリミティブな値や型のみで構成される命題を G' とする．パラメタ制約 C_{rsn} をこの様に特別に扱うのは，モデルパラメタがモデルの内部においては定数として働くためである．細分化モデルで利用されない定数をモデルパラメタとして持たないよう，パラメタ制約 C_{rsn} からの制約条件抽出では利用されないモデルパラメタについての制約をパラメタ制約から取り除く．

仮に I_{rsn} に対する出力を $G' = chs(G, vars(V'))$ と定義してしまうと G' に含まれる命題 G_i が X と X' 間の関係を定義するとき, 最弱事前条件 $[V']G_i$ が $[V]G_i$ と異なる命題になるため証明責務が偽になりうる．ただし, 初期化の代入中では変数を参照できず $vars(V') = X$ と $vars(V') - X' = vars(V')$ が成り立つため，V が初期化である場合には出力を $G' = chs(G, vars(V'))$ と定義しても証明責務は真である．提案手法では最弱事前条件をなす制約条件において V' で変化する変数 X と V で変化するが V' で変化しない変数 X' 間の関係を定義する命題を抽出しないことでこの問題を解決している．例えば, 不変条件として $W \subseteq ran(r)$ を持ち, 引数 w に対して $W := W \cup \{r(w)\}$ と $r := r \cup \{w \mapsto f(w)\}$ という 2 つの代入文を持つ操作を変数 W に関して細分化する事を考える．この時, 本手法に入力されるモデルの証明責務は真であるため, $W := W \cup \{r(w)\}$ || $r := r \cup \{w \mapsto f(w)\}$ という代入 V に対して $[V](W \subseteq ran(r))$ が真であることが保証される．操作分割では $W := W \cup \{r(w)\}$ という代入 V' が得られるが, この代入文が含む変数 r は細分化元の V で変化するため, $W \subseteq ran(r)$ は制約条件抽出において抽出しない．仮にこの制約を抽出すると最弱事前条件 $[W := W \cup \{r(w)\}]\ (W \subseteq ran(r))$ が真になるとは限らないため部品の信頼性を保証できない．

入力される制約条件として 5.3 節の制約条件展開が適用された制約条件を与えているが, これは変数間の関係の取りこぼしを解消するためである．入力モデルの制約条件には全ての変数間の関係が明示されてはおらず, 命題から推論される暗黙的な関係が存在する．この様な制約条件に対して上記のような命題の抽出を行うと変数間の関係に取りこぼしが生じるが, 5.3 節の制約条件展開で任意の変数間の関係が明示することによりこれを解消する．

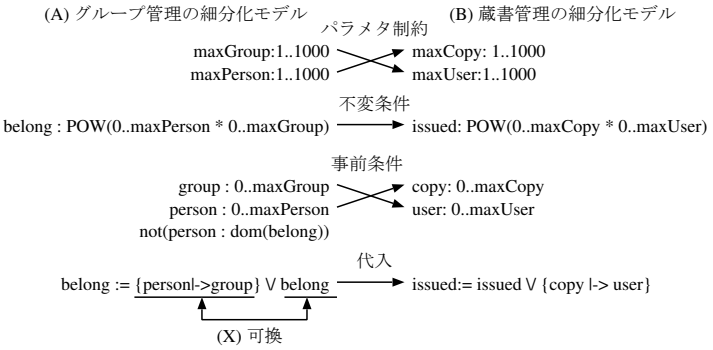


図 5.4: 構文要素整列例

5.6　構文要素整列

5.6.1　概要

構文要素整列は細分化モデルの構文要素を変数名に依存せず整列する．これは 6.2 節の部品検索において数学的同値判定と含意判定を定理証明によらず文字列一致で判定するためである．5.3 節の制約条件展開により数学的に等価な細分化モデルは同値な制約条件を持つため，同値な細分化モデルの字面が等しくなるには変数名に依らず構文要素を整列し，その後に変数名を出現順に統一すれば良い．例えば, 図 5.4 はグループ管理と蔵書管理の細分化モデル (抜粋) における各命題の対応関係を表したものである．この図では各命題間の論理積を省略して表現している．この例では矢印のように構文要素を並び替え，group を user に，person を copy に，belong を issued に置換することで事前条件以外の字面が等しくなる．部品検索における事前条件の扱いについては 6.2 節で詳しく説明するが，部品検索のマッチングでは事前条件以外の同値判定と事前条件の含意判定で行われる．構文要素の整列に必要な性質を以下に示す.

1. 整列結果は整列前の順不同な構文要素の並びに依存しない.
2. 整列結果は変数名に依存しない.
3. 同値判定の対象 (事前条件以外) の整列は含意判定の対象 (事前条件) に依存しない.

意味の等しい細分化モデル同士の整列結果が等しくなるには性質 (1)，(2) が必要である．また，含意判定の対象である事前条件に依存して同値判定の対象を整列した場合，事前条件のみが異なる細分化モデル間で文字列一致による同値判定が出来なくなるため性質 (3) が必要である.

本手法では構文要素に重みを定義し，構文木間で構文要素の重みを幅優先で比較することで構文木間に順序を与えて整列する．被演算式を入れ替えても意味が等しい演算子 (**可換演算子**) において被演算式間の順序を構文要素の重みから決定することで性質 (1) が得られる．この際，変数間の重みが等しいと被演算式間の順序を決定できない場合がある.

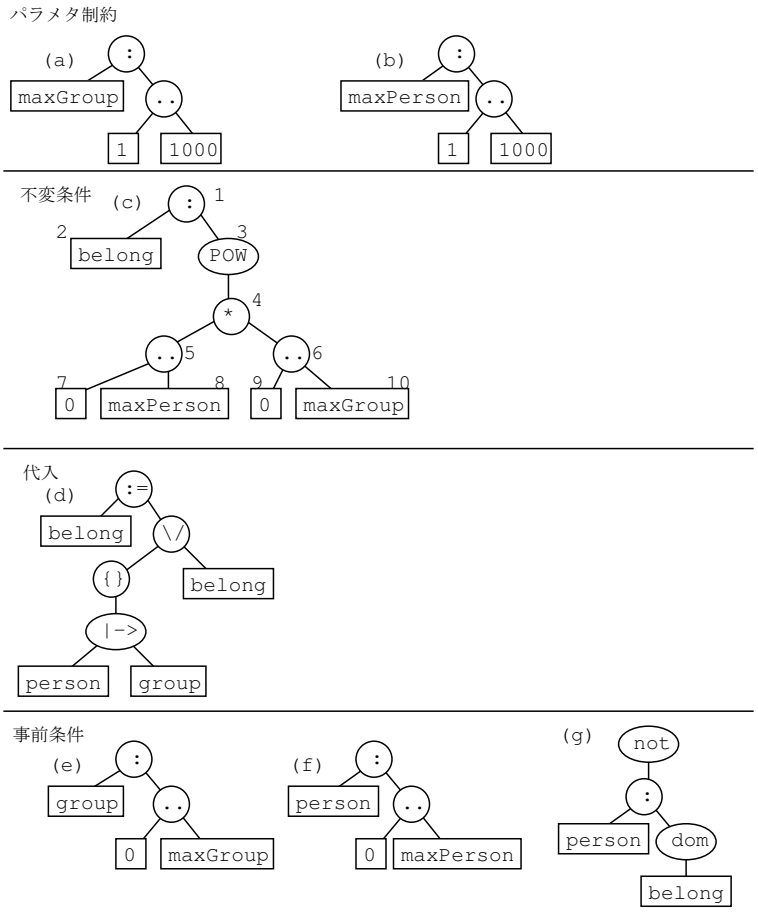


図 5.5: 構文木の例

そのため，本手法では推移律が成り立つ関係演算子から導出される変数間の半順序関係と決定済の構文要素の並びにおける変数の登場順から変数に重みを与える．これにより変数名に非依存に変数の重みが決定し性質 (2) が得られる．また，性質 (3) を得るために同値判定の対象であるパラメタ制約，プロパティ制約，不変条件，代入を整列した後に，そこで得られた変数の重みを用いて含意判定の対象である事前条件を整列する．これにより部品検索のマッチング (6.2.2 項) において同値判定が真であるときに限り，事前条件の並びが等しくなり含意判定を効率的に行える．以下に構文要素整列の手順を示す．

1. モデルの各式を構文木に変換する．モデルから構文木への変換は B Method における構文解析と同様に行う．図 5.5 は図 5.4(A) を構文木で表現したものである．例として図 5.5(a) は `maxGroup : 1..1000` を表す．
2. 変数間の重みを推移律が成り立つ関係演算子による半順序関係を用いて初期化する (**変数の初期重み付け**)．変数の初期重み付けは 5.6.2 項で説明する．
3. 可換演算子の被演算式の順序を統一する (**被演算式順序統一**)．被演算式順序統一は 5.6.3 項で説明する．

4. 木同士の構文要素の重みを評価してパラメタ制約, プロパティ制約, 不変条件, 操作の各構文木に順序をつける (**式順序付け**). なお, 細分化モデルの構文にはこの他に事前条件があるが, 事前条件の式順序付けはここで行わずに手順 (7) で行う. 式順序付けの詳細は 5.6.4 項で説明する.
5. 既知の式順序から変数の登場順序を求め, 各変数を順序付ける (**変数重み付け**). 変数重み付けの詳細は 5.6.5 項で説明する.
6. 構文木と変数の順序が新たに決定したら手順 (3) に戻る．また，構文木と変数の順序が新たに決定せず，重みが未決定な変数が存在する場合はそれらのうち一番重みが重い組 {a, b} において，$\mathrm{a} >_w \mathrm{b}$ とランダムに変数間に重みをつけ，重みが未定義な変数が存在しなくなるまで被演算式順序付け，式順序付け，変数重み付けを繰り返す.
7. 上記手順で求めた変数の重みで事前条件に被演算式順序統一と式順序付けを行う.

上記のように，新たに式順序が定まらなくなるまで被演算式順序付，式順序付け，変数重み付けを繰り返す．この繰り返しは式順序付けと変数重み付けが互いの結果を利用するためである．また，式順序が新たに決定せず，なおも，重みが未定義な変数が存在する場合には重みが未定義な変数の組 {a, b} において $\mathrm{a} >_w \mathrm{b}$ のようにランダムに変数間に重みをつける．このとき，$\mathrm{a} >_w \mathrm{b}$ としたときの整列結果と $\mathrm{b} >_w \mathrm{a}$ としたときの整列結果が等しい必要がある．これは変数 b と a を入れ替えても意味が等しい事を意味するが，これは 5.6.4 項の式順序付けで説明するように，その変数重みにおいて式順序が等しい式が等しい意味を持つ事から保証される.

以下の節では変数の初期重み付け，被演算式順序統一，式順序付け，変数重み付けについて説明する.

5.6.2　変数の初期重み付け

変数の初期重み付けでは関係演算子 $\leqslant$ あるいは $\in$ による半順序関係を用いて変数間の重みの大小を定める．制約条件は論理和を含む際には異なる半順序関係が複数得られる場合がある．この場合には最も多くの変数間の順序が決定する半順序関係を採用する．これも複数ある場合には，それらに共通する半順序関係のみを採用する．この様に半順序関係を採用した場合，変数間の重みの大小と変数間の論理的な大小関係が一致しない制約条件が生じるが整列に支障はない．これは変数の重みを決める際には ‘構文要素の並びと変数名のみが異なる細分化モデル間で変数の重みが等しく一意に定まる’ ことだけが必要だからである．なお，本手法では変数が重い式ほど先頭になるよう整列するため，$a \leqslant b$ の様な式に対して $a >_w b$ と重みを定める．この様に整列後の並び順を考慮して重みを定めることは必須ではないが 5.6.5 項の変数の重み付け結果との差異により混乱を生じな

いための考慮である．

初期重み付けでは $=, \leqslant, \in, \subseteq$ で定義される変数の大小，および包含関係から変数間に半順序関係を定める．ただし，プリミティブ化により $x \subseteq y$ は $x \in \mathbb{P}(y)$ に置換されている．関係演算子 $(=, \leqslant, \in)$ と変数および $\mathbb{P}$ のみで構成された式において，以下のように重みを定める．

1. $a = b$ ならば変数 a と変数 b の重みは等しい．
2. $a \leqslant b$，$a \in b$，$a \in \mathbb{P}(b)$ ならば変数の重みは $a >_w b$ である．

例えば，$a \leqslant c$, $a \leqslant b$, $b \leqslant c$ で定まる変数 a, b, c の重みは $a >_w b >_w c$ となる．変数の初期重み付けは式順序に依らず定まるため，モデルを構文木に変換した後に一度だけ行われる．

5.6.3　被演算式順序統一

可換演算子は被演算式が順不同な演算子である．被演算式順序統一は可換演算子の被演算式に対して式順序付けを適用し, 式順序で被演算式を整列する．ただし，被演算式の順序が決定出来ない場合は変数の重みが決定するまで整列しない．

図 5.4 では `\/`$(\cup)$ が可換演算子である．そのため，その被演算式である `{person |-> group}` と `belong` 間に式順序付けを適用し整列する．この例では変数 belong の方が集合演算子 `{}` より順序が早いため，`belong \/ {person |-> group}` と整列される．なお，図 5.4 は演算子 `*` $(\times)$ を含むが，これは集合の直積を表す演算子である．一般的に直積は $A \times B \neq B \times A$ の関係にあり，非可換演算子である．

5.6.4　式順序付け

式順序付けは式が属する節と式の構文要素で式間の順序を決定する．式が属する節による式順序付けではパラメタ制約, プロパティ制約, 不変条件, 代入の各節に属する式をこの順に順序付ける．節による順序付けは変数の登場順序を定めるために行う．変数の登場順序の決定に事前条件は用いないため, 節による順序付けは事前条件を含まない．構文要素による順序付けは構文要素の重みを幅優先で評価することで構文木間の順序関係を決定する．図 5.5(c) の各ノードの番号は構文木における評価順序を表す．構文要素の重みには表 5.3 を用いる．

構文木 T_A, T_B の要素の重みを幅優先で評価するとは次の手順を言う．T_A の i 番目の要素の親が T_A の k 番目の要素, T_B の i 番目の要素の親が T_B の l 番目の要素であるとする．この時, $k < l$ ならば $T_A > T_B$ である．$k = l$ であるとき, T_A の i 番目の要素が T_B の i 番目の要素より重いならば $T_A > T_B$ である．等しいならば $i + 1$ 番目の要素について同

表 5.3: 構文要素の重み

W	要素	W	要素	W	要素	W	要素	W	要素	W	要素
29	モデル引数	24	プリミティブな値	19	$\in$	14	$\mathbb{P}$	9	dom	4	id
28	モデル定数	23	関数呼び出し	18	$=$	13	$\{x\}$	8	; (合成)	3	$\forall$
27	モデル変数	22	$+$	17	$\leqslant$	12	..	7	$-$(差集合)	2	min
26	操作引数	21	$\times$	16	$\neg$	11	[]	6	$\times$(直積)	1	max
25	局所変数	20	/	15	$:=$	10	$\cup$	5	$^{-1}$(逆像)		

様の判定を行う. 全ての要素について T_A, T_B の順序が決定しないとき, $T_A = T_B$ である.

構造が等しく変数が異なる式を順序付けるため，変数毎の重みを 5.6.2 項の ‘変数の初期重み付け’ と 5.6.5 項の ‘変数重み付け’ で与える．例えば, 構文木中の変数重みが未定義のとき，図 5.5 の構文木の式順序は同順を () で表すと式の属する節から (a, b), c, d となる．さらに，変数の重みとして maxPerson $>_w$ maxGroup が与えられたとき, 式順序 b, a が決まる.

5.6.3 項の被演算式順序統一によって可換演算子の被演算式を式順序の早い順に並べることで可換演算子を意識せずに式順序の判定が行える．可換演算子の被演算式や変数の式順序が未定義の場合，どの様な並びであっても式順序付けの結果に影響しない. これは式順序が等しいならば, その時点で判明している変数の重みにおいて式の構造が等しい事を意味し, それらを入れ替えても式順序付けの結果は変わらないためである. 式順序付けと変数重み付けは互いの結果を用いる事で, それまで等しいと判定していた順序をさらに細かく順序付けする. そのため, 新たに順序が定まらなくなるまで式順序付けと変数重み付けを繰り返す.

5.6.5　変数重み付け

重みが未定義の変数について式順序付けで得られる式順で構文木を幅優先探索し，変数の出現順に変数が重いと定義する．ただし, 同順の式順序や重みの等しい可換演算子の被演算式に対してはそれらに同時に幅優先探索し, 同じ順序で出現した変数は次の出現が早い方を重いと定義する．例えば図 5.5 において (a, b), c, d という式順序であるとき, 式順序の早い順序に幅優先探索した時の変数の出現順序は (maxGroup, maxPerson), belong, maxPerson, maxGroup, ... である．ここで ( ) は同順で出現することを意味する. 同順である場合は, 次に現れる順序によって変数の重みを決めるため, maxPerson $>_w$ maxGroup の変数順序が定まる. これにより, 変数の重み maxPerson $>_w$ maxGroup $>_w$ belong $>_w$ person $>_w$ group が定まる.

5.7　モデル細分化手法の信頼性保証

高信頼な入力モデルを細分化したとき, 常に高信頼な細分化モデルが得られるならばその細分化手法は高信頼と言える．本節では整合性が保証されたモデル, すなわち, $C_A \wedge P_A \Rightarrow [U_A]I_A$ (式 2.4), $C_A \wedge P_A \wedge Q_{Aj} \wedge I_A \Rightarrow [V_{Aj}]I_A$ (式 2.5) が真であるモデルに対してモデル細分化を適用して得られる細分化モデルにおいて証明責務が真になることを証明し, これにより提案したモデル細分化手法の信頼性を保証する．細分化モデルの証明責務は式 (4.5) に示した初期化に関する証明責務 $C'_A \wedge P'_A \Rightarrow [V'_A]I'_A$, および, 式 (4.6) に示した操作に関する証明責務 $C'_A \wedge P'_A \wedge Q'_A \wedge I'_A \Rightarrow [V'_A]I'_A$ である．5.7.1 項では入力モデルの初期化を細分化して得られる細分化モデルにおいて初期化に関する証明責務が真になることを示す．また, 5.7.2 項では入力モデルの操作を細分化して得られる細分化モデルにおいて操作に関する証明責務が真になることを示す．ただし, 5.7.2 項に示すように操作に関する証明責務が真になるか否かは制約条件展開に依存するため, 得られた細分化モデルに対して人間が証明責務を証明しなければならない．この信頼性と人間の負担については 9.2.2 項で考察する．

5.7.1　初期化の整合性保証

2.5 節の式 (2.4) にモデルにおける初期化の証明責務を示した．この式においてパラメタ制約 C, プロパティ制約 P には不変条件とは異なりモデル変数が登場せず, パラメタとプロパティがどの様な型に属すといった制約のみが記述される．そのため, $C_A \wedge P_A$ が偽であるときを考慮する事は重要ではない．この初期化の整合性保証では $[U_A]I_A$ のみに注目する．細分化モデルの初期化と不変条件をそれぞれ U'_A, I'_A としたとき, 初期化に関するモデル細分化手法の信頼性を表す命題を式 (5.4) に定義する．式 (5.4) が成り立つ事を証明 1 に示す.

$$[U_A]I_A \Rightarrow [U'_A]I'_A \tag{5.4}$$

証明 1　式 *(5.4)* の証明

任意の命題 P, 任意の変数群 v について $P \Rightarrow chs(P, v)$ であることは明らかである．$[U_A]I_A$ は不変条件 I_A に対して代入文 U_A による変数置換を行った命題である．そのため, 任意の変数群 v について式 *(5.5)* が成り立つ．ここで注目する等価な変数群を ss としたとき $U'_A = slice(U_A, ss)$ であるため, 任意の変数群 v を $vars(slice(U_A, ss))$ とする事で制約条件抽出の定義から式 *(5.6)* が得られる．式 *(5.6)* の右辺の U_A のうち, 実際に変数置換を行っているのは I'_A が含む大域変数群に対して代入を行っているものだけである．ここで, $slice(U_A, ss)$ が置換する変数群 ss と I'_A が含む大域変数群 $vars(slice(U_A, ss))$ が等しいならば式 *(5.7)* が成り立つ．初期化では大域変数は代入文の左オペランドにしか現れないため, U'_A が含む大域変数と U'_A が置換する変数群は常に等しい．すなわち,

$vars(slice(U_A, ss)) = ss$ が成り立つ．よって，式 *(5.7)* が成り立つ．以上により，式 *(5.4)* が保証される．

$$
\begin{aligned}
[U_A]I_A &\Rightarrow [U_A]chs(I_A, v) && (5.5)\\
&\Rightarrow [U_A]chs(I_A, vars(slice(U_A, ss))) = [U_A]I'_A && (5.6)\\
&\Rightarrow [slice(U_A, ss)]chs(I_A, vars(slice(U_A, ss))) = [U'_A]I'_A && (5.7)
\end{aligned}
$$

5.7.2　操作の整合性保証

2.5 節の式 (2.5) にモデルの操作における証明責務を示した．この式のパラメタ制約 C とプロパティ制約 P を 5.7.1 項と同様の理由で省略し，$Q_A \wedge I_A \Rightarrow [V_A]I_A$ に注目する．細分化モデルの事前条件，不変条件，代入をそれぞれ Q'_A，I'_A，V'_A としたとき，操作に関するモデル細分化手法の信頼性を表す命題を式 (5.8) に定義する．

$$(Q_A \wedge I_A \Rightarrow [V_A]I_A) \Rightarrow (Q'_A \wedge I'_A \Rightarrow [V'_A]I'_A) \tag{5.8}$$

式 (5.8) が成り立つことを証明 2 に示す．この証明では任意の変数間の関係を命題 P から推論する推論器 $rsn(P)$ を想定する．5.3 節の制約条件展開がこの推論器の役割を担うが，実際には理想的な推論器を実装することは不可能である．そのため，操作の整合性保証は制約条件展開に依存する．

証明 2 式 *(5.8)* の証明

任意の変数間の関係を命題 P から推論する推論器 $rsn(P)$ を想定する．この推論器を用いたとき，$X \Rightarrow Y$ が真であり，ある変数群 v' に関して $X' = chs(rsn(X), v')$，$Y' = chs(rsn(Y), v')$ であるならば，$(X \Rightarrow Y) \Rightarrow (X' \Rightarrow Y')$ は真である．これは，推論器 rsn によって変数群 v' 間の関係が推論された事で，v' に関する命題 Y' を証明するのに必要な仮定 X' が得られるためである．ここで，X を $Q_A \wedge I_A$，Y を $[V_A]I_A$ に置き換える事では目的とする命題にはならないことに注意する．なぜなら，$[V'_A]I'_A$ が $[V_A]rsn(I_A)$ に存在しない命題を持つ場合には $Y' = chs(rsn(Y_A), v')$ の前提が崩れるからである．$[V'_A]I'_A$ が $[V_A]rsn(I_A)$ には存在しない命題を持つ場合とは，V_A では $y := y'$ と変化した変数 y が V'_A では $y := y$ となり変化せず，I'_A が変数 y を含む命題を持つ場合である．*5.5* 節の制約条件抽出において V_A では変化するが V'_A では変化しない変数を v' から除くことで $[V'_A]chs(rsn(I), v')$ が $[V_A]rsn(I_A)$ には存在しない命題を持たないようにしている．これにより，$Y' = chs(rsn(Y), v')$ の前提が成り立ち式 *(5.8)* が保証される．

第6章

モデル充足ソフトウェア合成

6.1 概要

本章ではモデル充足ソフトウェア合成 (Model Satisfiable Software Synthesis, MSSS) を提案する．MSSS は B Method のモデル (要求モデル) を入力として, 要求モデルに対する実装 (合成実装) と要求モデルに実装依存の制約を加えたモデル (合成モデル) を出力する. 例として要求モデルとそれに対する合成モデルと合成実装の例を図 6.1 に示す. 以下では要求モデル M_K を式 (6.1), 合成モデル M_A を式 (6.2) 合成実装 M_I を式 (6.3) のように表す.

$$M_K = (C_K, P_K, I_K, U_K, ((Q_{K1}, V_{K1}), \ldots, (Q_{Kl}, V_{Kl}))) \tag{6.1}$$

$$M_A = (C_A, P_A, I_A, U_A, ((Q_{A1}, V_{A1}), \ldots, (Q_{Al}, V_{Al}))) \tag{6.2}$$

$$M_I = (P_I, R_I, I_I, U_I, (V_{I1}, \ldots, V_{Il})) \tag{6.3}$$

出力モデル M_A は要求モデル M_K に対して実装に必要な記述を追加したモデルである．本手法では式 (4.3) の細分化モデルと実装依存モデル間の拡張の定義と同様に，要求モデルと出力モデル間に式 (6.4) が成り立つように合成する．式 (6.4) は合成モデルの制約条件が要求モデルに実装依存の制約条件を加えたものであり，初期化と操作の代入が等しいことを意味する．式 (6.4) には事前条件についての記述が無いが，合成モデルの事前条件 Q_{Aj} は Q_{Kj} と同値またはより弱い制約に実装依存の制約を加えた命題である．これは要求モデルの事前条件 Q_{Kj} が‘その条件において代入 V_{Kj} が実行できる’という要求を表すためである．

$$(C_A \Rightarrow C_K) \wedge (P_A \Rightarrow P_K) \wedge (I_A \Rightarrow I_K) \wedge (U_A = U_K) \wedge (V_{Aj} = V_{Kj}) \tag{6.4}$$

MSFC の信頼性の定義である充足と同様に，式 (6.4) に加えて出力モデル M_A と出力実装 M_I 間で式 (2.4), 式 (2.5), 式 (2.6) の証明責務群を生成し, それらを証明することでソフトウェアの信頼性を保証する. 本研究では式 (6.4)，式 (2.4), 式 (2.5), 式 (2.6) が成り立

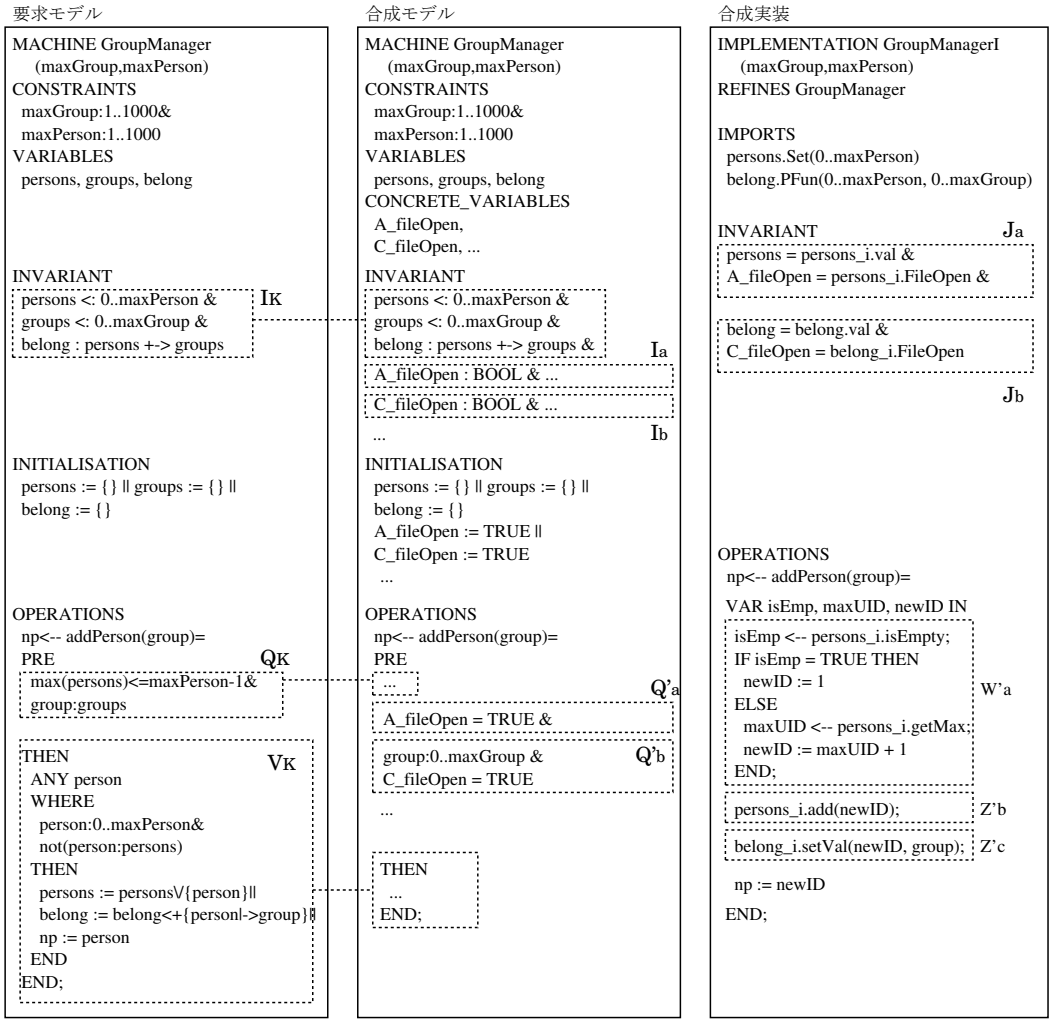


図 6.1: MSSS によるソフトウェア合成の入出力例

つことを**モデル充足**と呼び，合成手法の信頼性の定義とする．この信頼性を満たす合成を実現するために，本研究では要求モデルを細分化した検索キーによる健全性の高い部品再利用を提案する．また，MSFC および 部品検索を数学的に定義することで，合成手順で得られるソフトウェアの信頼性を定性的に評価し，モデル充足となる為に必要な人間の証明作業を明確にする．

図 6.2 に示すように MSSS の手順は大きく分けてモデル細分化，部品検索, 変数名統一, 部品選択, 部品結合からなる. 以下に各手順を簡単に説明する．

1. **モデル細分化:** 要求モデルを機械的に細分化して 4.2.1 項で定義した細分化モデル群を生成する．細分化モデルは部品の仕様であるため，これを検索キーとしてリポジトリ内の細分化モデルを検索することで，小さな計算量で健全性の高い部品検索ができる．
2. **部品検索:** モデル細分化で得られた検索キーにより部品群を検索する. 詳細は 6.2 節で述べるが，得られる部品が検索キーと同じ機能を有すること，また，検索キーで与えられた条件下で実行可能な事を数学的な命題として与える．これを定理証明に基づいて数学的に判定することで検索の健全性と完全性を保証できる.
3. **変数名統一:** 検索で得られた部品の変数名は部品間で異なるため，合成時に要求モデルを満たすように検索キーの変数名を用いて変数名を統一する．
4. **部品選択:** 要求モデルから得られた細分化モデル毎にたかだか 1 つの部品を選択する. これは図 6.2 のように 1 つの細分化モデルに対して複数の部品が検索で得られるためである. この際, I/O などの数学的には記述できない要求に応じて部品を選択するため, 部品の記述を読解すること無く部品を選択できる仕組みを提案する.
5. **部品結合:** 部品の実装依存モデルを結合することで合成モデルを作り, 部品の細分化実装を結合することで合成実装を作る.

MSSS の信頼性については 9.2.1 項で詳しく考察するが，要求モデルが実装依存の制約条件を含まないならばモデル充足の式 (6.4) を満たす合成モデルが得られる．また，要求モデルが無矛盾であり，再利用した部品間に矛盾が無いならば，式 (2.4), 式 (2.5), 式 (2.6) の合成モデルの無矛盾性，及び合成モデルと実装間の整合性を保証できる．合成モデルの不変条件について簡単に説明すると，要求モデルの不変条件 I_K，その検索キーの不変条件 I_{Ki}，に対して，モデル細分化を $I_K \Leftrightarrow \bigwedge I_{Ki}$ と定義し，I_{Ki} と同値な不変条件を持つ部品を検索するためである．これにより得られる部品の実装依存モデルの不変条件 I_{Di} は 4.2.2 項に示したように，$I_{Ai} \Rightarrow I_{Ki}$ であるため，$\bigwedge(I_{Ai} \Rightarrow I_{Ki}) \Rightarrow (\bigwedge I_{Ai} \Rightarrow \bigwedge I_{Ki})$ より $I_{Ai} \Rightarrow \bigwedge I_{Ki}(\Leftrightarrow I_K)$ となる．‘要求モデルが実装依存の制約条件を含まないならば’ という条件は細分化モデルは実装依存の制約を持たない為に生じる条件である．

部品検索では定理証明に基づき数学的に判定することで健全性と完全性を保証できる．しかし，MSSS では膨大な部品群を持つリポジトリを想定するため，部品数だけ定理証明

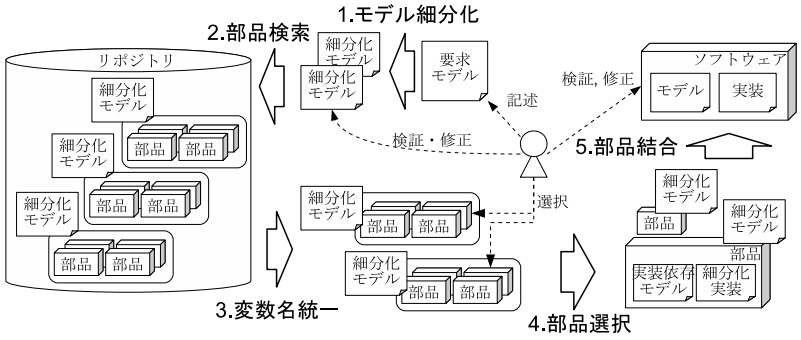


図 6.2: モデル充足ソフトウェア合成の流れ

を繰り返すことは現実的ではない．そこで，本研究では数学的に等価な細分化モデルの字面が等しくなるようにモデル細分化を構築し，部品数だけ繰り返す判定を文字列一致で行うことを提案する．

得られた合成モデルと合成実装を B Method の枠組みで検証し，証明責務が真であるならば ComenC[8] などのコード生成器を用いて実行可能コードを得る．証明責務が偽になる理由として‘部品選択時に部品を割り当てなかった細分化モデルが存在する’，‘選択した部品間に矛盾する記述が存在する’ ことが挙げられる．この様な場合には利用者による部品の追加や部品選択のやり直しが必要となる．MSSS では以下の作業を利用者が行う．

1. 要求モデルの記述
2. 細分化モデル群の無矛盾検証，および，矛盾時の制約条件の追加
3. 部品選択における実装方法の選択
4. 成果物に対する証明責務の証明，および，証明失敗時の修正

MSSS は検証作業や実装方法の選択などを検索時に行うため，既存の自動コード生成よりも作業量が多いように見える．しかし，検証作業により成果物の信頼性を保証できるためデバッグの手間を省くことが出来，また，手戻りを抑えることが出来るため，既存手法より作業量が多いとは言えない．MSSS は利用者による実装方法の選択が必要であるが，MSSS の入力は DSL などの既存手法より抽象的であり, 既存手法においても仕様記述時に実装方法を記述するため, MSSS の作業量が既存手法より多いとは言えない．

モデル細分化は 5 章で説明した．部品検索，部品選択，部品結合についてはそれぞれ 6.2 節, 6.3 節, 6.4 節で説明する．なお，変数名統一は部品検索時に行うため，6.2 節で合わせて説明する．合成されたソフトウェアの証明責務が偽である場合の対応については 6.5 節で説明する．また，モデル細分化の信頼性については 5.7 節で説明する．

6.2　部品検索

6.2.1　概要

部品検索は図 6.2 に示すように要求モデル M_K から得られたある細分化モデル M'_K に対して部品群 $(G_i, 1 \leqslant i \leqslant n)$ を出力する．ここで, G_i は細分化モデル M'_K を満たす仕様を持った部品群である．これにより, 検索で得られた部品を組み合わせたソフトウェアが要求モデルを満たす事が期待できる．部品検索の出力において要求モデルから得られる細分化モデル M'_K を $(C'_K, P'_K, I'_K, Q'_K, V'_K)$, 部品の細分化モデル M'_A を $(C'_A, P'_A, I'_A, Q'_A, V'_A)$ としたとき, M'_A が M'_K を満たすとは M'_A の変数名を置換したときに M'_A と M'_K の間で式 (6.5) が成り立つことを言う．本稿では M'_A が M'_K を満たすことを $M'_A \subseteq_S M'_K$ と表現する．

$$((C'_K \wedge P'_K \wedge I'_K) \Leftrightarrow (C'_A \wedge P'_A \wedge I'_A)) \wedge (V'_K = V'_A) \wedge (Q'_K \Rightarrow Q'_A) \tag{6.5}$$

‘変数名を置換したとき式 (6.5) が成り立つ’ とは数学的に記述を評価することを意味する．例えば, 数式 E において y という変数名が使われていないならば, E の変数 x を全て y に書き換えても意味は等しい．この様に数学的な仕様を解釈する際には変数名の同異は重要だが, 変数名は意味を持たない．そのため, M'_A の変数名を M'_K の変数名に合わせて置換して細分化モデルを数学的に比較する．変数名を置換した際に式 (6.5) を満たすとは細分化モデル M'_A が検索キー M'_K と同じ機能を持ち, M'_K の代入文が実行可能な条件において M'_A も実行可能であることを意味する．ただし, 式 (6.5) の判定には定理証明を要し, この判定を部品数だけ繰り返すことは現実的でない．本手法では計算量を低減するために検索キー M'_K と部品の細分化モデル M'_A の字面を構文要素整列 (5.6 節) で統一することで，文字列一致で判定する．

部品検索のマッチングでは検索キー M'_K に対して $M'_A \subseteq_S M'_K$ を満たす細分化モデル M'_A を部品リポジトリから検索し, 細分化モデル M'_A を仕様として持つ部品群 G_i を出力する．細分化モデル間のマッチングの詳細は 6.2.2 項で説明する．

命題を文字列一致で判定するには ‘推論による命題の統一’ と ‘命題の並びの統一’ が必要である．‘推論による命題の統一’ は数学的に等しい命題群から変数名と構文要素の順序だけが異なる命題群を推論により導出する．さらに, ‘命題の並びの統一’ により構文要素の並びを統一して命題を文字列一致で判定する．このうち ‘推論による命題の統一’ はモデル細分化の ‘制約条件展開 (5.3 項)’ で，‘命題の並びの統一’ はモデル細分化の ‘構文要素整列 (5.6 項)’ で行われる．構文要素整列では 2 変数が全く同じ制約を持つ場合に構文要素の並びが非決定的になるが, この問題は判定結果に影響しない．これは, 全く同じ制約を持つ変数は変数名を互いに入れ替えても意味が変化しないためである．MSSS ではモデルが数学的に等しくても制約条件展開で得られる記述が変数名と構文要素の順序以外に差異を持つ場合に ‘数学的に等しくない’ と誤判定する．詳しくは 9.3 節で考察するが,

制約条件展開の推論には大きな計算量を必要とするため, 検索の完全性と計算量のトレードオフを考慮して制約条件展開を整備する必要がある.

6.2.2　マッチング

本節では細分化モデル間における文字列一致による式 (6.5) の判定をしめす．以下に検索キー M_K と細分化モデル M_A 間の判定手順を示す.

1. 事前条件以外の制約条件と代入について M_A の変数を登場順に M_K の変数名に置換する．ここで決定した変数名置換を M_A の事前条件にも適用する．
2. M_K と変数名を置換した M_A の事前条件以外の制約条件と操作の字面が完全一致しないならば判定結果は偽である．
3. 論理積を区切りとして命題を分割したとき，変数名を置換して M_A の命題を M_K が全て含むならば判定結果は真である．すなわち，M_A の命題数を n_A，M_K の命題数を n_K としたとき，$n_K < n_A$ ならば偽である．また，5.6.1 項冒頭で示した構文要素整列の性質 (3) により事前条件において M_A の各命題の登場順序は M_K の事前条件における登場順序と等しい．よって，M_A と M_K の事前条件の各命題の対応は M_A の各事前条件と字面が一致する事前条件を M_K から登場順に探索することで n_K 回の判定で得られる．

図 6.3 は検索キーとそれにマッチする細分化モデル，および部品を示す．ここでは図 6.3 の検索キー c と細分化モデル c 間のマッチング手順を例示する．細分化モデル c の変数は登場順に X1, X2, X3, X4, X5 である．これを検索キー c の変数に置換すると，それぞれ maxPerson, maxGrp, belong, person, group となる．この時，事前条件以外の制約条件と操作の字面は完全一致する．次に，事前条件の命題数は検索キー c で 3, 細分化モデル c で 2，よって $n_K \geqslant n_A$ となる．この 2 つの細分化モデル c の事前条件は検索キーに含まれるため，マッチング結果は真である．

手順 (1) の変数名のつけ替えで未決定の変数名が生じるのは M_K と M_A で登場する変数の数が異なる場合である．この場合には操作か不変条件に差異があるため手順 (2) で同値関係にならない．他にも，操作や不変条件に登場しない変数が事前条件に登場する場合にはその変数の置換を手順 (1) で決定できない．しかし，細分化モデルの定義から事前条件や不変条件に登場する変数は操作に登場する変数とその変数を説明するのに必要な変数であるため，不変条件や操作に登場しない変数が事前条件に登場することはない．このため，手順 (2) で同値関係になる細分化モデルは手順 (1) で全ての変数が置換される．式 (6.5) の数学的判定は事前条件以外の同値関係と事前条件の含意関係からなる．手順 (2) により事前条件以外の制約条件における同値関係を文字列の完全一致で判定する．また，$Q_K \Rightarrow Q_A$ という含意関係は Q_K, Q_A を論理積で区切られた命題を要素とした集合と捉え

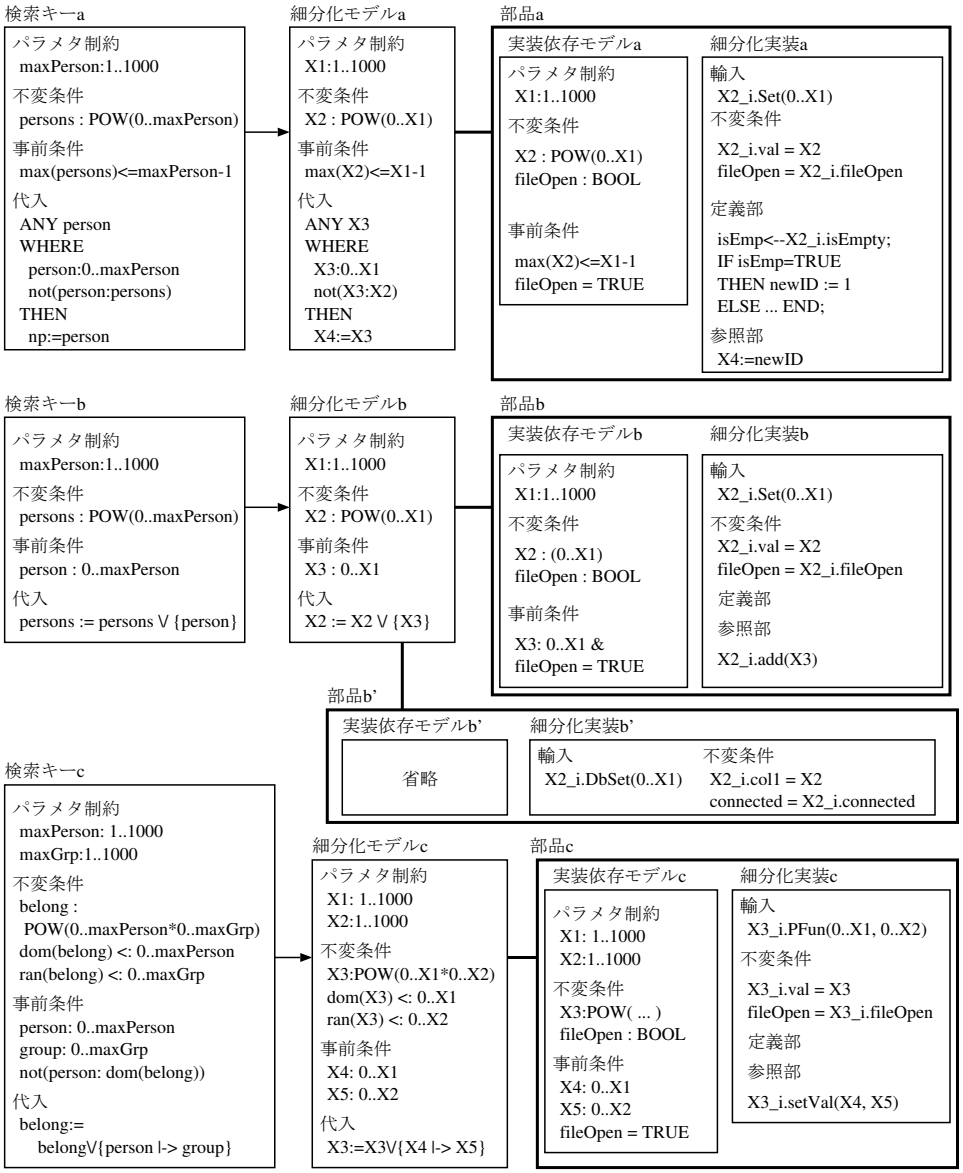


図 6.3: 検索で得られる部品群

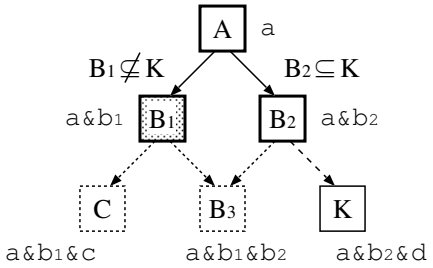


図 6.4: 階層構造を利用した探索空間の限定

ることで，$Q_A \subseteq Q_K$ という要素の包含関係で表される．これにより，Q_A の変数名を Q_K の変数名に置換することで字面一致で判定できる．

この判定を部品リポジトリに格納された各細分化モデルに対して行うが，4.4.2 項の部品リポジトリの階層構造により探索空間を限定する．この探索空間の限定手順は ‘検索キーにマッチしなかった細分化モデルの子孫を探索しない’ というルールにより行われる．細分化モデルが検索キーにマッチしないという事は ‘事前条件以外が同値でない’ か ‘検索キーにない命題が事前条件に存在する’ 事を意味する．事前条件以外の制約条件が検索キーと同値でない細分化モデル M_A，M_A の子孫 M_{Ai} としたとき，M_{Ai} と M_A の事前条件以外の制約条件は同値であるため，M_{Ai} の事前条件以外の制約条件も検索キーと同値にならない．このため，M_{Ai} は検索キーにマッチしない．また，検索キーにない命題 p を事前条件に持つ細分化モデル M_B，M_B の子孫 M_{Bi} としたとき，M_{Bi} も命題 p を持つため，M_{Bi} は検索キーにマッチしない．

例えば，図 6.4 の様な細分化モデル A, B_1, B_2,B_3, C が格納された部品リポジトリに対して命題 a, b_2, d で構成された事前条件 $a \wedge b_2 \wedge d$ を持つ検索キー K で検索することを考える．上記の判定手順 (3) では $a \Leftarrow a \wedge b_2 \wedge d$ という含意関係は $\{a\} \subseteq \{a, b_2, d\}$ という包含関係で表される．このため，K が含まない命題 b_1 を持つ細分化モデル B_1 は K にマッチせず，その子孫である C, B_3 もマッチしない．

6.2.3　変数名統一

変数名統一は部品群の変数名を要求モデルの変数名に統一し，結合時に部品群を連動させる．この処理は検索キー M'_K と, それに割り当てられた部品 G を入力として, M'_K の変数 x_i に対応する部品 G の変数 y_i を特定し, y_i の変数名を x_i に合わせる．

以下の節では変数名統一を実装依存モデルの変数名統一と細分化実装の変数名統一に分けて説明する．

実装依存モデルの変数名統一

実装依存モデルの変数名統一では実装依存モデル群の変数名を検索キー，すなわち，要求モデルの変数名に置換する．6.2 項の部品検索時に式 (6.5) が真になるよう部品の細分化モデルの変数名を検索キーの変数名で置換し，それと同様に実装依存モデルの変数名を置換することで行う．実装依存変数群には部品間での参照や代入を避けるために固有の変数名を与える．

例として図 6.3 の各部品に対する変数名置換の結果を図 6.5 に示す．ここでは検索キー a, b に対する変数名置換を説明する．細分化モデル a の変数名を登場順にそれぞれ検索キーの変数名に置換すると X1, X2, X3, X4 がそれぞれ maxPerson, persons, person, np に置換される．同様に細分化モデル b では X1, X2, X3 がそれぞれ maxPerson, persons, person に置換される．よって実装依存モデルの変数に対しても同様の置換を適用する．また，部品 a, b の実装依存変数 fileOpen は A, B をそれぞれ接頭語とし，部品に固有の変数名 A_fileOpen, B_fileOpen を与える．

細分化実装の変数名統一

細分化実装の変数名統一では部品結合時に連動して動作するように，細分化実装の変数名を実装依存モデルの変数名に合わせて置換する．モデル実装間変数対応付け (4.4.3 項) により実装依存モデルのモデル変数と細分化実装の大域変数は 1 対 1 に対応する．ここではモデル変数に対応する実装変数を **リンク変数** と呼び，モデル変数を用いてリンク変数の変数名を統一することを考える．実装変数は具象変数，輸入変数，局所変数に分類でき，それぞれ以下の様に置換する．この際，‘輸入変数の置換’ ではモジュール別名を共通化し，部品結合時に部品群を協調させる．また，リンク変数が等しい実装依存変数名を共通化することで，実装依存変数を要求モデルで共通化できない問題を解消する．

1. **具象変数の置換:** モデル変数 x_{Ki} のリンク変数 v_i が具象変数ならば v_i を x_{Ki}_i に置換する．
2. **輸入変数の置換:** モデル変数 x_{Ki} のリンク変数 v_i がモジュール M で宣言される場合，輸入 ‘別名. モジュール M’ を ‘x_{Ki}_i. モジュール M’ に置換する．また，モジュール変数等の参照も同様に置換する．複数の実装変数が 1 つのモジュールで宣言される場合，適当な実装変数 y を 1 つ選び，他の変数名は y と同様に置換する．この際，リンク変数が等しい全実装依存変数に共通の変数名を与える．
3. **局所変数の置換:** 細分化実装の局所変数には部品間で重複しない変数名を与える．本稿では重複する局所変数名に部品名を付加する．

モデル実装間変数対応付けはモデル変数と実装変数の対であり，図 6.3 の 部品 a, b では共に (X2, X2_i.val), (fileOpen, X2_i.fileOpen) と定まる．実装依存モデルの変数名統一

図 6.5: 変数名置換を適用した部品群

では X2 を persons に置換したため，モジュールの別名 X2_i を persons_i に置換する．図 6.3 の部品 a, b においてリンク変数が等しい実装依存変数としては fileOpen が挙げられる．このため，B_fileOpen を 部品 A に合わせて A_fileOpen に置換する．局所変数の置換では部品間で局所変数名の重複が生じないため，isEmp などの局所変数名をそのまま用いる．これにより，図 6.5 のように変数名置換後の部品が得られる．

6.3　部品選択

6.3.1　概要

部品選択は図 6.2 の様に複数の部品を持つ細分化モデル群を入力とし, たかだか 1 つの部品を持つ細分化モデル群を出力する．部品選択の出力において部品を持たない細分化モデル M_K が存在すると，合成して得られる出力ソフトウェアは M_K に対する実装を持たない．この様な不完全なソフトウェアへの対処は 6.5 節で説明する．部品選択に望ましい性質を以下に示す．

計算量 合成ソフトウェアの証明責務が真になる部品の組を選択したい．これは組み合わせ問題となるため，計算コストの小さな判定方法が望ましい．

完全性 利用可能な部品を選択不可と誤判定することによる既存部品の再実装の手間を避けるため，完全性を保証することが望ましい．

作業量 細分化モデルが同一でもプラットフォームなどが異なる部品が有るため，利用者が部品を選択する．しかし，部品の読解は困難なため容易な選択手法が望ましい．

計算量を小さくし，また完全性を保証するために合成ソフトウェアの証明責務が偽になる部品の組合わせを判定する選択可否判定を提案する (6.3.2 項)．また，作業量を軽減するために部品を読解せずに目的にあった部品を選択する手法を提案する (6.3.3 項)．

6.3.2　選択可否判定

選択可否判定は部品の組に対してその組が選択可能か否かを出力する．部品の組が選択不可能とは，それらを結合したソフトウェアの証明責務が偽になることを意味する．これにより，合成後の証明責務が偽になることによる手戻りを軽減する．しかし，部品選択時に証明責務を保証することは，膨大な部品の組合わせに定理証明を適用するため困難である．また，定理証明では証明失敗も ‘真でない’ と判定されるため，真である証明責務を偽と誤判定する可能性がある．実際，B Method の自動定理証明器では条件分岐や繰り返しを含むソフトウェアに対しては人手による証明を必要とし，証明責務の自動

定理証明で選択可否判定を行うと，それらを再利用できない．提案手法ではこの様な誤判定による再利用性の低下の回避と計算量軽減のため，証明責務が偽になる事が明らかな組を型で判定する．この手法では合成後に証明責務が偽になる場合が生じうるが，この問題については 6.5 項で対応する．

部品 A, B 間の選択可否判定は以下のように行う．部品 A, B の実装依存モデルに共通する全てのモデル変数について，変数名統一 (6.2.3 項) で得られたリンク変数の型が一致するならば選択可否判定は真である．モデル変数 x の部品 A, B のリンク変数をそれぞれ v_A, v_B とする．この時，v_A と v_B の型が一致するならば選択可否判定は真である．これは v_A と v_B は共にモデル変数 x を表す実装変数であり, 合成された実装で v_A, v_B は同じ 1 実装変数になるためである．型の一致判定は以下の (1), (2), (3) の判定が全て真である時に限り真である．

1. v_A, v_B の一方が輸入変数，一方が具象変数である場合, 選択可否判定は偽である．
2. v_A, v_B が輸入変数であり，それぞれ異なるモジュールで宣言される場合, 選択可否判定は偽である．
3. v_A, v_B が具象変数であり，その型が不変条件において排他的であるならば選択可否判定は偽である．変数 v の型は不変条件において $v \in X$ や $v \subseteq Y$ と定義されるプリミティブな集合 X, Y である．型 X_1, X_2 が排他的とは $X_1 \cap X_2 = \emptyset$ と同意である．

図 6.3 では検索キー a に対して部品 a が，検索キー b に対して部品 a, b' が得られる．ここでは部品 a と 部品 b, b' 間の選択可否判定を例示する．部品 a, b 間では検索キーの変数 persons のリンク変数はともにモジュール Set で定義される X2_i.val であり，型が一致する．同様に共通する全てのモデル変数についてリンク変数の型が一致するため，選択可否判定は真である．一方，部品 b' ではモデル変数 person のリンク変数がモジュール DbSet で定義される X2_i.col1 であり，部品 a と型が一致しない．このため，部品 a, b' 間の選択可否判定は偽である．

6.3.3　利用者による実装方法の選択

MSSS では利用者の要求をモデルに記述するが, モデルは抽象度が高く対象とするプラットフォームや I/O の実装方法を特定できないため，部品選択により実装方法を選択する．しかし, 部品の読解は困難であるため, ‘要求モデルの変数を実装するモジュール’ の選択により実装方法を選択する．以下に選択手順を示す．

1. モデル変数に対して選択可能なモジュールの候補を抽出し，利用者に提示する．これは 6.2.3 項の変数名統一時に ‘輸入変数の置換’ で特定される．
2. 利用者がモデル変数 x_{Ki} に対してモジュール M を選択する．3.3.1 項で述べたように,

モジュールは概念ごとに構築されるため，モデル変数で表したい概念に合致するモジュールを選択する．なお，実装方法を配慮しないモデル変数に対してはモジュールを選択をしない．

3. モデル変数 x_{Ki} を含む検索キー群の部品候補をモジュール M で x_{Ki} のリンク変数を宣言する部品群に限定する．
4. 6.3.2 項の選択可否判定を適用し，手順 2 に戻る．

B Method の実装記述には I/O に関する単語が存在せず, I/O はモジュールを用いて記述する．そのため, モデル変数を実装するモジュールを選択することで部品の記述を読解せずに実装方法を選択できる．例えば, 6.3.2 項で例題に用いたモジュール DbSet と部品 b' を例にすると, 部品 b, b' はモデル変数 persons のリンク変数をそれぞれモジュール Set, DbSet を輸入して実装している．このため，利用者による実装方法の選択では persons に対するモジュールの候補として Set, DbSet が提示される．これらは集合を永続化するモジュールであり，Set はファイルで集合を扱うための，DbSet はデータベースで集合を扱うためのモジュールである．モデル変数 persons をファイルで永続化したい場合はモジュール Set を選び，このとき部品 b' が出力から外れ，結果として部品 a, b の組み合わせが選択される．逆に，DbSet を選んだ場合，部品 a, b が出力から外れ，部品 b' が選択される．なお，この場合，細分化モデル a に対する部品が存在しないため，部品追加が必要となるが，これについては 6.5 項で説明する．

6.4　部品結合

6.4.1　概要

k 個の細分化モデル M'_{Ki}, $(1 \leqslant i \leqslant k)$ に対して部品選択で選んだ部品の実装依存モデルと細分化実装をそれぞれ $M'_{Di} = (C'_{Di}, P'_{Di}, I'_{Di}, Q'_{Di})$, $M'_{Ii} = (P'_{Ii}, R'_{Ii}, I'_{Ii}, W_i, Z_i)$ とする．部品結合はこれらの部品群と要求モデル $(C_K, P_K, I_K, U_K, ((Q_{K1}, V_{K1}), \ldots, (Q_{Kk}, V_{Kl})))$ を入力として合成モデル $(C_A, P_A, I_A, U_A, ((Q_{A1}, V_{A1}), \ldots, (Q_{Ak}, V_{Al})))$ と合成実装 $(P_I, R_I, I_I, U_I, (V_{I1}, \ldots, V_{Il}))$ を出力する．6.1 節で述べたように, 合成モデルは要求モデルに対して $C_A \Rightarrow C_K$, $P_A \Rightarrow P_K$, $I_A \Rightarrow I_K$, $U_A = U_K$, $V_{Ai} = V_{Ki}$ を満たす必要がある．また，合成実装は合成モデルに対する詳細化であり, 合成モデルと合成実装から生成した証明責務は真になることが期待される．

部品結合手順を以下に示す．

1. **輸入:**　実装依存モデルの輸入 R_{Ii} を重複を省き全て列挙して R_I を得る．
2. **制約条件:**　合成モデルのパラメタ制約 C_A, プロパティ制約 P_A, 不変条件 I_A を $C_A = C_K \wedge \bigwedge_{i=1}^{k} C_{Di}$ のように要求モデルと実装依存モデルの制約条件の論理積で

得る．出力実装のプロパティ制約 P_I, 不変条件 I_I も同様に実装依存モデルの制約条件の論理積で得る．
図 6.5 の部品 a, b の実装依存モデルの不変条件を I_a, I_b, このとき, 部品 a, b を結合した図 6.1 の合成モデルの不変条件は $I_K \wedge I_a \wedge I_b$ である．また，部品 a, b の細分化実装の不変条件を J_a, J_b とすると部品 a, b を結合して得られる合成モデルの不変条件は図 6.1 の様に $J_a \wedge J_b$ である．

3. **実装操作:**　出力モデルの事前条件 Q_{Aj}，出力実装の代入 V_{Ij} を 6.4.2 項の ‘操作の合成’ により得る．同様の手順で実装の初期化 U_I も得る．
4. **不完全なソフトウェアへの対応:**　6.3.2 項で述べたように MSSS の出力は証明責務が偽になりうる．この場合の対応を 6.5 項に示す．

モデル充足の定義 (6.1 節) を満たす合成モデルの制約条件，すなわち，$C_A \Rightarrow C_K$, $P_A \Rightarrow P_K$, $I_A \Rightarrow I_K$, となる合成モデルの制約条件は実装依存モデルの制約条件を論理積で結合することで得られる．これは部品の定義と検索の定義から保証される．まず，要求モデル M_K から得られる細分化モデルを M_{Ki}, それを検索キーとして得られる部品の細分化モデルと実装依存モデルを M_{Ai}, M_{Di} とする．また，M_{Ki}, M_{Ai}, M_{Di} の不変条件を I_{Ki}, I_{Ai}, I_{Di} とする．要求モデル M_K が実装依存の制約を含まないとき，$I_K = \bigwedge I_{Ki}$ が成り立つ．検索の定義から $I_{K_i} \Leftrightarrow I_{Ai}$ であり，部品の定義より $I_{Di} \Rightarrow I_{Ai}$ が成り立つ．よって，$I_{Di} \Rightarrow I_{Ki}$ であり，この論理積を考えると $\bigwedge(I_{Di} \Rightarrow I_{Ki}) \Rightarrow (\bigwedge I_{Di} \Rightarrow \bigwedge I_{Ki})$ より，$\bigwedge I_{Di} \Rightarrow I_K$ が得られる．この事から合成モデルの制約条件を実装依存モデルの制約条件の論理積とする事で，その制約条件はモデル充足の定義を満たす．ただし，要求モデルが実装依存の制約条件を含む場合を考慮し，ここでは出力モデルに要求モデルの制約条件を論理積で加えている．

以下の節では操作の合成と不完全なソフトウェアへの対応について説明する．

6.4.2　操作の合成

要求モデルの操作 (Q_K, V_K) から得た細分化モデル群で検索された部品群の実装依存モデルの事前条件を Q_{Di}，細分化実装の代入を (W_i, Z_i) とする．操作の合成は (Q_K, V_K), Q_{Di}，(W_i, Z_i) を入力として出力モデルの事前条件 Q_A と出力実装の代入 V_I を出力する．出力実装の代入 V_I は要求モデルのモデル細分化と逆の手順で部品群の代入 (W_i, Z_i) を結合することで得る．図 5.1 のようにモデル細分化では操作を非決定的値生成操作 V_{Dec} と非決定的値参照操作 V_{Ref} に分割し，それらを大域変数と操作戻値への代入文 V_{Ai} 毎に分割した．これを逆に結合するには，V_{Ai} を同時代入で結合して V_{Dec} と V_{Ref} をそれぞれ再構築し，それらを結合することで細分化元モデルの操作 V_A を得る．この結合を実装で行うには ‘逐次代入による結合時の副作用解消’ と ‘V_{Dec} から V_{Ref} への値の引き渡し’ を考

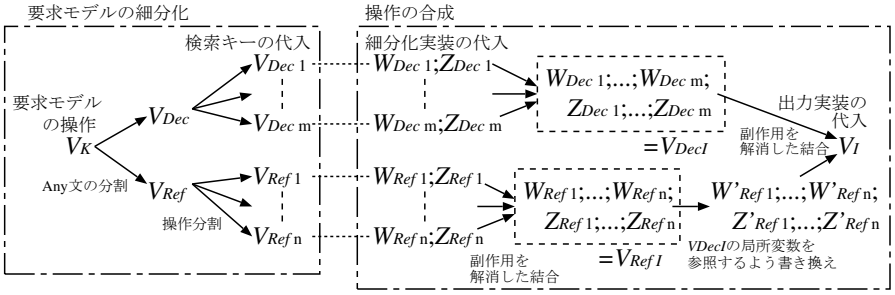


図 6.6: 操作の合成の流れとモデル細分化との対応

える必要がある．副作用解消方法は 4.2.3 項で説明した．値の引き渡しでは変数の書き換えにより V_{Dec} で定義した局所変数を V_{Ref} に引き渡す．また，事前条件は不変条件と同様に論理積で結合できるが，非決定的値参照操作の事前条件に ANY 文の WHERE 節を論理積で追加しているため，これを取り除く．

出力実装の代入 V_I は以下のように定義される代入 $W_{Dec\,p}$, W'_{Refq}, Z'_{Refq} に対して式 (6.6) で表される．図 6.6 は式 (6.6) に現れる要素の導出過程と要求モデルに対するモデル細分化との対応を表す．図 6.6 において $V_{Dec\,p}$, $(1 \leqslant p \leqslant m)$ と V_{Recq}, $(1 \leqslant q \leqslant n)$ はそれぞれ要求モデルの非決定的値生成操作 V_{Dec} と非決定的値参照操作 V_{Rec} を操作分割した代入群である．また，$V_{Dec\,p}$ で検索された実装依存モデルの事前条件を $Q_{Dec\,p}$，細分化実装の代入を $(W_{Dec\,p}, Z_{Dec\,p})$ とする．同様に V_{Refq} で検索された実装依存モデルの事前条件を Q_{Refq}, 細分化実装の代入を (W_{Refq}, Z_{Refq}) とする．さらに，非決定的値生成操作の代入部 $W_{Dec\,p}$ において戻値 res_x に代入される式が E であるとき，(W_{Refq}, Z_{Refq}) において変数 x を式 E に置換した代入を (W'_{Refq}, Z'_{Refq}) とする．この時，出力実装の操作 V_I は式 (6.6) である．また，出力モデルの事前条件 Q_A は $(\bigwedge Q_{Dec\,p}) \wedge (\bigwedge Q_{Refq})$ から V_I で宣言される変数を含む命題を取り除いたものである．V_I で宣言される変数を含む命題とは Q_{Refq} が含む局所変数に対する制約である．

$$V_I = W_{Dec\,1};\ \ldots;\ W_{Dec\,m};\ W'_{Ref1};\ \ldots;\ W'_{Refn};\ Z'_{Ref1};\ \ldots;\ Z'_{Refn} \tag{6.6}$$

図 6.6 において V_{Dec} は $V_{Dec\,1} \,||\, \ldots \,||\, V_{Dec\,m}$ であるため, これと同様に $V_{Dec\,p}$ に対する実装の代入同士を結合することで V_{Dec} に対する実装の代入 V_{DecI} が得られる．4.2.3 項で示したように実装操作をモデル操作と同様に結合するには参照部 W が代入部 Z の前に来るよう逐次代入で結合する．これにより代入 V_{DecI} は $W_{Dec\,1};\ \ldots;\ W_{Dec\,m};\ Z_{Dec\,1};\ \ldots;\ Z_{Dec\,m}$, 代入 V_{RefI} は $W_{Ref1};\ \ldots;\ W_{Refn};\ Z_{Ref1};\ \ldots;\ Z_{Refn}$ になる．これらを結合して V_I を得るため，(W_{Refq}, Z_{Refq}) から (W'_{Refq}, Z'_{Refq}) への書き換えにより，V_{DecI} で定義した局所変数を V_{RefI} に引き渡す．(W'_{Refq}, Z'_{Refq}) は $Z_{Dec\,p}$ を介さずに $W_{Dec\,p}$ で定義した値を参照するため $Z_{Dec\,p}$ は V_I に不要である．以上により式 (6.6) が出力実装の代入となる．

ここでは図 6.5 の部品 a, b を再利用して図 6.1 の合成モデルの事前条件と合成実装の操作を合成する例を示す．合成モデルの事前条件は結合する部品 a, b の実装依存モデルの

事前条件 Q_a, Q_b を変数名置換し, 要求モデルの操作 V_A 内で宣言される変数 person を含む命題を取り除いた命題をそれぞれ Q'_a, Q'_b とする．この時, 部品 a, b を結合した合成モデルの事前条件は図 6.1 の様に $Q_K \wedge Q'_a \wedge Q'_b$ となる．次に合成実装の操作を合成する．非決定的値生成操作に対する部品は部品 a だけであるため, 式 (6.6) の $W_1^d;\ \ldots;\ W_{m_d}^d$ にあたる代入文は部品 a の参照部 W_a である．また, 部品 b は参照部を持たず, 定義部 Z_b だけを持つ．Z_b の変数 user を式 newID で置換した代入文を Z'_b としたとき, Z'_b は式 (6.6) の $Z_1'^r;\ \ldots;\ Z_{m_r}'^r$ の一部である．

6.5　不完全なソフトウェアへの対応

6.5.1　証明責務が偽になる原因と対応

MSSS では合成されたソフトウェアの証明責務が偽になる可能性がある．証明責務が偽の場合には合成されたソフトウェアに問題があるため，これに対処する必要がある．なお，証明責務の証明は B Method の枠組みを利用して人手で行う．MSSS で証明責務が偽になる原因を以下に挙げる．

1. 要求モデルには存在しないモデル変数が合成モデルに追加された．
2. 部品選択で選択した部品が互いに矛盾する記述を持つ．
3. 6.3 節の部品選択で部品の割り当てがない細分化モデルが存在する．

原因 1 は実装依存モデルにより要求モデルに存在しないモデル変数が追加される事で初期化されない変数が生じるため証明責務が偽になる．原因 2 は 6.3.2 項で提案した選択可否判定で証明責務が偽になる部品の組を真と誤判定した際に生じる．原因 3 のように部品が割り当てられない部品が存在する場合には証明を行うまでもなく，合成ソフトウェアの証明責務が偽になることが明らかである．以下の節ではこれらの原因それぞれについて修正方法を述べる．

6.5.2　実装依存変数が追加された場合

合成モデルに要求モデルに無いモデル変数 X が追加された場合，以下のように出力モデルを修正し，これを要求モデルとして再度 MSSS を適用する．

1. モデル変数 X の初期化を追加する．
2. モデル変数 X に対する操作を必要に応じて追加する．
3. 初期化，および追加した操作の証明責務を証明する．

修正 1, 3 は合成モデルが要求モデルにない変数を持つ場合には必ず必要である．修正 2 では実装方法に応じて ‘ファイルを開く’, ‘ファイルを閉じる’ などの操作を加える．なお，この様な操作を追加しなくても合成で得られる操作群に矛盾は生じない．しかし，事前条件が常に偽である操作が存在したり，システム全体で見たときに仕様に欠落が生じる場合がある．これは，B Method では不変条件と事前条件に対して各操作が実行できる事のみを検証するためである．このため，どの順番で各操作を行っても，事前条件が真にならない操作が存在したり，本来あるべき ‘ファイルを閉じる’ 操作が存在しないのような仕様の欠落を検証できない．

上述の修正を行った後，以下のように MSSS を再度適用する．この際，モデル細分化と部品選択は前回の結果を再利用できる．

1. 追加した変数 X に注目して初期化にモデル細分化を適用する．
2. 追加した操作にモデル細分化を適用する．
3. 1, 2 で得られた細分化モデルを検索キーとして部品群を得る．その他の検索キーに対しては前回の検索結果，および部品選択をそのまま利用する．
4. 追加された検索キーで得られた部品群に前回の部品選択結果を用いて選択可否判定を適用する．
5. 選択された部品群に部品結合を適用する．

6.5.3　選択した部品が互いに矛盾を持つ場合

選択した部品が互いに矛盾する場合，部品選択に誤りがあるため，部品選択をやり直す．部品選択に誤りが生じるのは，‘部品選択時の選択可否判定が偽である’ ことと ‘証明責務が偽である’ ことが同値でないためである.. この判定精度の低下は計算量の観点から証明責務が偽になることが明らかな部品のみを選択不可と判定しているためである．

例として，モデルにおいて $users \subseteq \mathbb{N}$ と与えられたモデル変数 users の実装変数 users_i に対する実装の制約条件を考える．部品 A の実装の不変条件は `min(users_i.val) > 0` という制約条件を持ち，部品 B は users_i の要素に 0 を追加する代入を持つとする．この時，部品 A, 部品 B は共にモデルを満たすが，部品 A, 部品 B を組み合わせると，部品 B の代入結果が部品 A の不変条件に違反する．なお，部品 A では実装の不変条件がモデルの不変条件より厳しい制約を持つが，証明責務の観点からは問題ない．これは B Method ではモデルの事前条件が真ならば操作を実行できる必要があるが，実装の不変条件がモデルの不変条件を網羅することは求められていないためである．

部品の再選択時に部品選択の誤りによる合成失敗を繰り返さないためには，証明責務を偽にする制約条件を特定し，これを含む部品を選択しない必要がある．矛盾する制約条件は証明責務の証明時に特定できる．このため，部品選択の際，矛盾する制約条件を

含む部品群を除外することで合成失敗の繰り返しを回避できる．部品選択後は再度，部品合成を適用して合成ソフトウェアの証明を行う．上述の例では `min(users_i.val) > 0` という不変条件を満たす事を証明するのに失敗するため，この不変条件を持たない部品を部品 A の代わりに再利用すればよい．

6.5.4　選択できる部品が存在しない場合

選択できる部品が存在しない場合には実装に必要な部品が不足しているため，この部品を利用者が追加する．部品の追加では他の部品の記述と矛盾しないように足りない機能の細分化実装を記述し, 証明責務が真になるように実装依存モデルに制約を追加する．ただし, 合成された合成実装は可読性が低いため, 利用者が合成実装の記述を読まずに他の部品と整合性を持つ部品を記述できるように以下の補助を行う．

1. **部品の仕様の提示:** 要求モデルから生成される細分化モデルが部品の仕様に当たるため, これを提示することで実装すべき機能を明確化する．
2. **類似部品の提示:** 6.2.2 項で提案したマッチングは高い健全性を持つ反面，利用できる部品を取りこぼす恐れがある．このため，類似部品として検索キーと事前条件のみが異なる部品群を提示する．この際，部品の事前条件から検索キーが持たない事前条件を取り除く．提示された部品に対して利用者は実装依存モデルと細分化実装の整合性を定理証明により検証し，これが真であるばあいは，その部品を利用する．本研究では検索キーの事前条件より厳しい事前条件を持つ部品を ‘実行できない可能性がある’ として再利用しないため，この様な取りこぼしが生じる．また，事前条件は厳しければ厳しい程，モデルと実装間の整合性の証明責務が真になるため，部品の汎用性を向上するには不要な事前条件を取り除く必要がある．例えば，図 6.3 の部品 c は登録済の要素に対しても実行可能だが，開発時には検索キーのようにより厳しい制約を事前条件を与えられる．この場合，それらの事前条件を取り除き，必要以上に厳しい事前条件を与えていないかを証明により検証することで部品の汎用性を向上できる．
3. **リンク不変条件とモジュールの輸入の自動生成:** 他の部品群で定めた変数のリンク不変条件と利用モジュールは変更できないため, リンク不変条件とモジュールの輸入を自動生成する．
4. **実装依存変数と制約条件の自動生成:** 実装依存モデルを記述する際, 他の部品で定めた実装依存変数の型は変更できないため, 型を定義する不変条件を自動生成する．また, 証明責務を真にするには輸入モジュールの事前条件を守る必要があるため, 輸入モジュールに合わせて事前条件を追加する．この事前条件の追加では輸入モジュールの事前条件が等しい部品の実装依存モデルの事前条件が参考になる場合がある．その

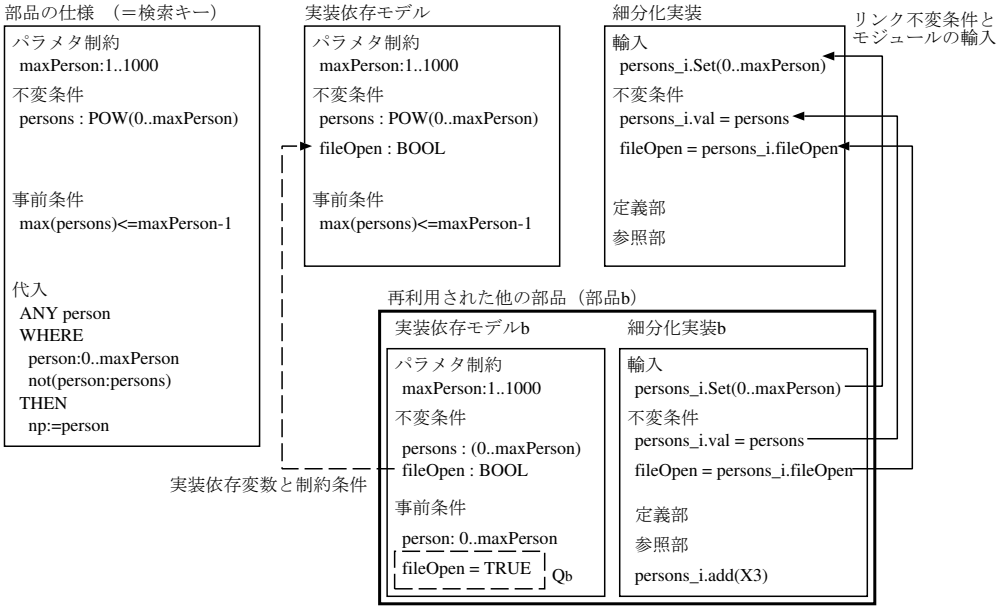


図 6.7: 部品追加時の提示物

ため, 輸入モジュールの事前条件が等しい部品が存在する場合にはその部品の実装依存モデルの事前条件を利用者に提示する.

例として図 6.3 の部品 a がリポジトリに存在せず検索キー a に部品が割り当てられない場合を考える. この場合, 作業者は上述の補助を受けて実装依存モデルと細分化実装を記述する. 補助 1 では足りない部品の仕様として図 6.7 のように検索キーが部品の使用として提示される. 図 6.3 において細分化モデル a が含み要求モデルが含まない変数は np だけであり, これも操作の戻値である事が容易に分かるため, 要求モデルを記述した利用者であれば細分化モデル a の読解は容易である.

次に補助 3 では他の検索キーに割り当てられた部品の細分化実装から追加部品の細分化実装の雛形を提示する. この例では検索キー b に対して部品 b が割り当てられているため, 変数 persons に対応する実装変数が `persons_i.Set` に限定される. これによりモジュールの輸入とリンク不変条件が自動生成され, 図 6.7 の細分化実装が提示される.

最後に補助 4 では実装依存モデルと細分化実装間の証明責務が真になるよう, 部品の細分化モデルに実装依存の制約を追記することで実装依存モデルを記述する. この例では実装依存変数 fileOpen の型は部品 b の fileOpen と同じであるため, 不変条件として fileOpen:BOOL が自動生成される. また, 部品 a と部品 b で呼び出すモジュール操作の事前条件がともに `fileOpen=TRUE` を持つため, 部品 b の事前条件 Q_b を事前条件の候補として提示する. 以上により図 6.7 の実装依存モデルが提示される. この例では事前条件 Q_b を実装依存モデルに追記するだけで部品 a の証明責務が真になる.

第7章

モデル充足細粒度部品生成

7.1 概要

本章ではB Method のソフトウェア から MSFC を生成するモデル充足細粒度部品生成 (MSFC 生成) を提案する．MSFC 生成は入力されるソフトウェアの証明責務が真であるならば，得られる部品の証明責務も真であることを保証する．MSSS を含む部品再利用による自動コード生成手法は部品が整備された問題領域にしか適用できない．この問題に対して MSFC 生成を利用することで，その問題領域に既存のソフトウェアが存在すればそこから部品を自動生成できるため，自動コード生成手法の適用範囲を広げることができる．また，一般的に高信頼なソフトウェア部品の整備には大きなコストを要するが，MSFC 生成では高信頼なソフトウェアを容易に得られる．

図 7.1 に示すように MSFC 生成はモデル細分化，実装抽出，部品登録からなる．モデル細分化により入力ソフトウェアのモデルから細分化モデル群を生成し，実装抽出により細分化モデルを ‘充足する’ 部品を入力ソフトウェアから生成する．さらに，部品登録により得られた部品を部品リポジトリに登録する．3.1 節で定義したように，部品が細分化モデルを充足するとは実装依存モデルが細分化モデルの拡張であり，細分化実装と実装依存モデル間で証明責務が真になる事を表す．6.2 節の部品検索では部品の仕様である細分化モデル M_A が細分化モデル M_K を ‘満たす’ ことを $M'_A \subseteq_S M'_K$ と定義した．6.2 節

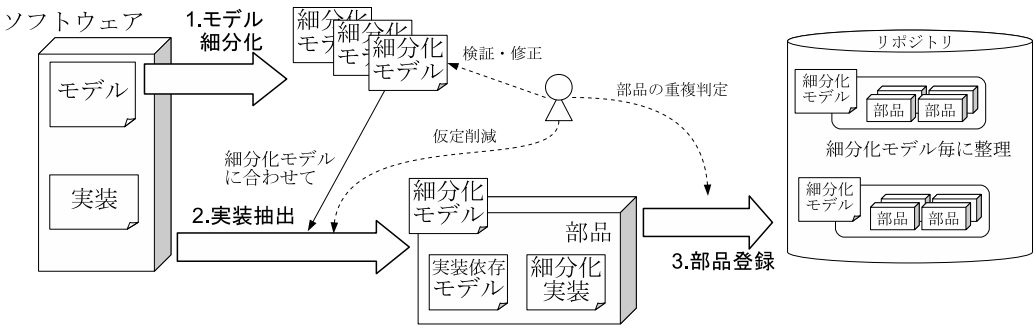


図 7.1: モデル充足細粒度部品生成の流れ

で定義した ‘満たす’ と 3.1 節で定義した ‘充足する’ は異なる概念である．これは，‘満たす’ は実装依存の表現を持たない細分化モデル間の関係であるのに対して，‘充足する’ は実装依存の表現を持つ部品と実装依存の表現を持たない細分化モデル間の関係である事からも明らかである．

MSFC 生成においてモデル細分化は MSSS で要求モデルから細分化モデル群を生成するのに用いたモデル細分化と同じ処理である．これにより，MSSS と MSFC 生成間で部品の粒度が等しいことを保証できる．モデル細分化で生成される細分化モデルが無矛盾である事が求められたように，実装抽出では生成される実装依存モデルと細分化実装が細分化モデルを充足する事が求められる．5.7 節に示したようにモデル細分化では細分化手順を定め，その手順で得られる記述が証明責務を満たす事を証明により示した．しかし，細分化実装に関する証明責務は細分化モデルのものに比べて複雑であるため，モデル細分化と同様の方法では部品が細分化モデルを充足することを保証することは困難である．そこで MSFC 生成では記述を証明責務に変換し，証明責務が真であることを保ちながら証明責務を最小化する事でこれを保証する．部品登録では 4.4.2 項で示したリポジトリの構造を保つよう部品をリポジトリに格納する．部品リポジトリでは部品は等価な細分化モデル毎にまとめて保存されるため，MSSS の部品検索で定義した細分化モデル間の数学的判定を利用して部品の細分化モデルと等価な細分化モデルをリポジトリから検索する．実装抽出については 7.2 節で説明する．部品登録については 7.3 節で説明する．MSFC 生成の実例は 8.3 節で示すが，6 章で用いた図 6.3 の部品群は簡単な図書館システムに MSFC 生成を適用して得られた部品群である．

7.2　実装抽出

7.2.1　概要

実装抽出は細分化モデルに合わせて実装から細分化実装と実装依存モデルを抽出する．本章では実装抽出の入力細分化モデルを式 (7.1)，入力モデルを式 (7.2)，入力実装を式 (7.3) の様にあらわす．

$$M'_A = (C'_A, P'_A, I'_A, Q'_A, V'_A) \tag{7.1}$$

$$M_A = (C_A, P_A, I_A, Q_A, V_A) \tag{7.2}$$

$$M_I = (P_I, R_I, I_I, V_I) \tag{7.3}$$

実装抽出では細分化モデル M'_A の細分化元操作以外の操作は関与しないため，式 (7.2) と式 (7.3) のように入力モデルと入力実装から関与しない操作を取り除いている．入力モデルと入力実装からなるソフトウェアは事前に信頼性が保証されているとする．すなわち，入力モデルと入力実装において式 (2.6) に示した実装の操作に関する証明責務は真である．

この入力に対して出力される実装依存モデルを式 (7.4)，細分化実装を式 (7.5) の様にあらわす.

$$M'_D = (C'_D, P'_D, I'_D, Q'_D, V'_D) \tag{7.4}$$

$$M'_I = (P'_I, R'_I, I'_I, W, Z) \tag{7.5}$$

7.1 節で述べたように出力される MSFC は入力細分化モデルを充足しなければならない. 充足は ‘実装依存モデルが細分化モデルの拡張である事’ と ‘実装依存モデルと細分化実装で証明責務が真である事’ で表される．そのため，実装依存モデルの代入 V'_D は拡張の定義から細分化モデルの代入 V'_A そのものである．また，拡張の定義から C'_D, P'_D, I'_D, Q'_D は細分化モデルの制約と実装依存の制約の論理積である．実装依存モデル M'_D と細分化実装 M'_I において式 (4.7) の証明責務が真になることを保証することで部品の信頼性が保証される. また, 実装依存モデルが細分化モデルの拡張となるため部品が仕様を満たすことが保証される. 以下に実装抽出手順を示す.

1. **大域変数の対応付け:** モデルに合わせて実装を抽出するためにモデルの大域変数に対応する実装の大域変数を抽出する. 詳細は 7.2.2 項で説明する.
2. **非決定的値生成の分離:** モデル細分化の ‘非決定的値生成の分離’ で得られる非決定的値生成操作, 非決定的値参照操作に対応するように実装の操作を分割する. 詳細は 7.2.3 項で説明する.
3. **操作抽出:** 2.4.2 項で紹介した Weiser の手法を応用して実装操作から細分化モデルの操作と等価な操作を抽出する. 詳細は 7.2.4 項で説明する.
4. **副作用解消:** 実装の代入を大域変数を参照する参照部と大域変数を変更する代入部に分割することで副作用を解消する. 詳細は 7.2.5 項で説明する.
5. **証明責務最小化:** 副作用解消で得られた細分化実装の操作に対して式 (4.7) の証明責務が真になる制約条件を求める. 証明責務最小化で得られた制約条件から細分化実装の制約条件と実装依存モデルの制約条件を得る. 詳細は 7.2.6 項で説明する.

証明責務が真になる出力を得るため, 本研究では入力された記述から証明責務を生成し, その証明責務が真になるよう証明責務を最小化し, その証明責務から細分化実装と実装依存モデルを生成する事を考えた. これは, 手順 (5) によって証明責務のゴールを真にするような仮定を得ることで行う. そのため, ゴールを構成する細分化実装の代入 V'_{ref}; V'_{def} を先に定める必要がある. モデル細分化の粒度は ‘1 操作における 1 変数の状態変化’ であるため, 細分化モデルで変化する変数に対応する実装変数を y としたとき, 細分化実装の代入は実装操作末尾における変数 y の状態を計算するのに必要なプログラムであることが分かる. これを抽出するのが手順 (1) と手順 (3) である. ただし, モデル細分化では細粒度化のために非決定的値生成の分離 と SELECT 文の分割 を行ったため, 手順 (2) で実装操作についても非決定的値生成の分離を行うとともに, 操作抽出時に細分化モデルの制御構造に合わせて実装の制御構造を抽出する. 以下の節では実装抽出の各手順を説明する.

7.2.2　大域変数の対応付け

大域変数の対応付けはモデルの大域変数 x と実装を入力として実装の変数群 $Y = \{y_1, \ldots, y_k\}$ を出力する．ここで, 実装の変数群 Y は モデル変数 x を実装する変数群である. B Method ではモデルの変数は抽象変数と呼ばれ実装で扱えないため, 実装では実装変数を定義して実装変数と抽象変数の関係を不変条件に記述する．この様な不変条件をリンク不変条件と呼ぶ. 本稿では 3.3.2 に示したようにリンク不変条件を ‘抽象変数 = (実装変数の式)’ の形に制限している．そのため, 変数群 Y とプリミティブな値と型のみで構成された式を $E(Y)$ としたとき, 抽象変数 x に関するリンク不変条件は $x = E(Y)$ と表され, これにより実装においてモデル変数 x の代りに用いられる変数群 Y が決定する. 以降では変数群 Y をモデル変数 x のリンク変数と呼ぶ.

7.2.3　非決定的値生成の分離

概要

本節では 5.2 節で行ったモデルに対する ‘非決定的値生成の分離’ と同様の分離を実装に対しても行う．実装における非決定的値生成の分離はモデルにおける非決定的値生成操作の代入 $V_{AnyDefA}$ と非決定的値参照操作の代入 $V_{AnyRefA}$, 及び, 実装の代入 V_I を入力として実装における非決定的値生成操作 $V_{AnyDefI}$ と非決定的値参照操作 $V_{AnyRefI}$ を出力する．ここで $V_{AnyDefI}$, $V_{AnyRefI}$ はそれぞれ $V_{AnyDefA}$, $V_{AnyRefA}$ に対応する実装の操作である．この分割ではモデルの ANY 文で生成される値を格納する局所変数 v に対応する式 E を実装の操作から特定し, その式 E の値を生成する操作 $V_{AnyDefI}$ と式 E を用いて代入を行う操作 $V_{AnyRefI}$ に分割する．モデルと実装間の局所変数の対応づけはモデルにおける局所変数の参照と同様の参照が行われる式を実装から抽出することで行う．モデルの局所変数に対応する実装の式が特定出来たら, 非決定的値生成操作と非決定的値参照操作を構築する. 非決定的値生成操作の構築にはプログラムスライシングを用いる．また, 非決定的値参照操作では局所変数に対応する式の値を操作引数として受け取り, 局所変数に対応する式を引数で置換する.

非決定的値に対応する式の特定

細分化モデルの ANY 文において非決定的値 v が生成されるとする．非決定的値に対応する式の特定では細分化モデルの非決定的値 v に対応する実装の式 E を特定する．本手法では大域変数および戻値への代入文を手がかりに式 E を特定する．これは ANY 文から局所変数 v を用いた代入文を抽出し, それと同様の代入を行う代入文を実装から特定することで行う.

細分化元のモデルにおいて 1 大域変数への代入文が 1 操作に 1 つだけならばこの代入文の特定は容易である．しかし, 実際にはモデルと実装は条件分岐によって 1 大域変数に対して複数の代入文が記述されるため, モデルの代入文に対応する実装の代入文を特定するためには実装の制御構造とモデルの制御構造の対応を特定しなければならない．これは実装の条件式をリンク不変条件から抽象変数に書き換えた上で, モデルの条件式と比較することで行う．特定されたモデルと実装の代入文はそれぞれモデルの大域変数への代入文とそのリンク変数への代入文となっている．ここで, この 2 つの代入文は等価な代入文であるため, 代入文それぞれの右オペランドを等号で結んだ式が成り立つ．この式を変形してモデルの非決定的値 v に対して $v = E$ の形に等式を変形することで v に対応する式 E を特定する．

非決定的値生成，参照操作の構築

モデルの局所変数の値に対応する式 E を特定したら, ‘式 E を決定し, それを戻値として返す非決定的値生成操作’ と ‘式 E の値を引数として受け取り, それを用いて代入を行う非決定的値参照操作’ を構築する．実装の操作の戻値と引数は対応するモデルの操作のそれと等しくなければならない．そのため，実装の非決定的値生成操作 $V_{AnyDefI}$ と非決定的値参照操作 $V_{AnyRefI}$ の戻値と引数はそれぞれモデルにおける非決定的値生成操作 $V_{AnyDefA}$ と非決定的値参照操作 $V_{AnyRefA}$ のそれと等しい．

非決定的値生成操作 $V_{AnyDefI}$ は先の ‘非決定的値に対応する式の特定’ で特定された非決定的値に対応する実装の式 E を戻値に代入する操作である．そのため，非決定的値生成操作 $V_{AnyDefA}$ の戻値を v_{res} としたとき，$V_{AnyDefI}$ は細分化元の実装の代入 V_I において式 E を含む大域変数への代入の直前に代入文 $v_{res} := E$ を追加したものである．これだけでは $V_{AnyDefI}$ は E を決定するのに利用しない代入を含むが，そのような不要な代入はこの後に行う操作抽出においてプログラムスライシングによって取り除かれる．

非決定的値参照操作 $V_{AnyRefI}$ は式 E を含む大域変数への代入文において式 E を引数 v に置換した代入である．このため式 E の値の決定は不要であり，$V_{AnyRefI}$ は計算に寄与しない代入を含む．この様な不要な代入は操作抽出でプログラムスライシングによって取り除かれるため，$V_{AnyRefI}$ を定める際には取り除く必要がない．

7.2.4　操作抽出

操作抽出は細分化モデルの代入 V'_A と実装の代入 V_I を入力として V'_A と等価な実装の代入 V'_I を出力する．ここで, V'_A が非決定的値生成操作を細分化して得られる代入ならば V_I は実装の非決定的値生成操作 $V_{AnyDefI}$ であり, V'_A が非決定的値参照操作を細分化して得られる代入ならば V_I は実装の非決定的値参照操作 $V_{AnyRefI}$ である．この抽出にはプロ

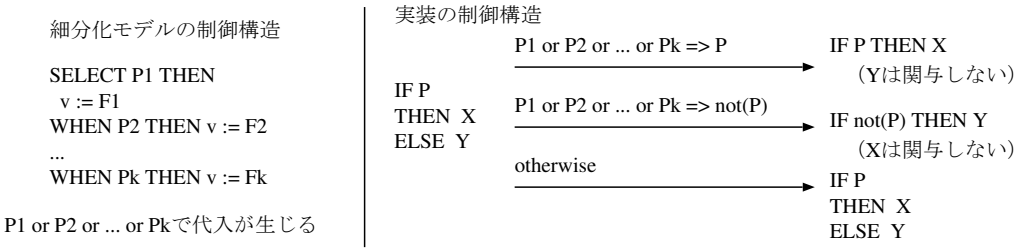


図 7.2: 実装の制御構造の抽出

グラムスライシング手法である Weiser の手法を用いる．モデル細分化では条件分岐を分割したが, Weiser の手法のスライシング結果は細分化モデルに含まれない遷移条件や代入を含む．本手法ではこの様な細分化モデルに反する実装を抽出しないためにモデル細分化で抽出した遷移条件に対する実装の抽出 (制御構造の抽出) とモデル細分化で抽出しなかった代入の特定 (抽出しない文の特定) を行う．この結果に対してプログラムスライシングを適用することで細分化モデルの代入 V'_A に対する実装を V_I から抽出する．以下の節では ‘制御構造の抽出’, ‘抽出しない文の特定’, ‘プログラムスライシングの適用’ について説明する．

制御構造の抽出

制御構造の抽出は細分化モデルにおける遷移先が空でない遷移条件群 $P_1, \ldots, P_k$ と実装の代入文 V_I を入力として, $P_1, \ldots, P_k$ に対応する遷移をのみを実装した実装の代入文を出力する．ここでは条件式が P である IF 文について THEN 部を条件式 P の制御ブロック, ELSE 部を条件式 $\neg P$ の制御ブロックと呼ぶ. 制御構造の抽出は実装の IF 文において条件 $P_1, \ldots, P_k$ で実行されない制御ブロックを空にすることで行う．すなわち，図 7.2 のように $(P_1 \vee \cdots \vee P_k) \Rightarrow P$ ならば条件式 $\neg P$ の制御ブロックを空にし，$(P_1 \vee \cdots \vee P_k) \Rightarrow \neg P$ ならば P の制御ブロックを空にする．また，どちらでも無い場合，すなわちどちらの制御ブロックも実行しうる場合には両方の制御ブロックを残す．これは細分化モデルで空でないブロックに到達する条件において到達不可能な制御ブロックを実装操作から取り除くことを意味する．

抽出しない文の特定

細分化モデルの代入 V'_A と V'_A に対応する細分化実装 V'_I を考える．V'_A で変化する変数群を X_A, V'_I における X_A のリンク変数を X_I とする．また, V'_A で変化しない変数群を Y_A, V'_I における Y_A のリンク変数を Y_I とする．この時, 細分化モデルの代入 V'_A と細分化実装の代入 V'_I が対応するためには, V'_A における X_A の変化と同様に V'_I において X_I が変

化することが重要であるが, V'_A において Y_A が変化しないのと同様に V'_I において Y_I が変化しないことも重要である. このため, Y_I は ‘変化してはいけない大域変数 (代入禁止実装大域変数)’ であり, Y_I を変化させないよう細分化実装の操作 V'_I を生成しなければならない. これは代入禁止実装大域変数群 Y_I への代入文 f にマークを付け，後に行うプログラムスライシングにおいてマークが付いた代入文を抽出しない事で実現する. また, 代入文 f が操作呼び出しである場合, 1 代入文で複数の値が変化する場合がある. このように 1 代入文で代入禁止実装大域変数 Y_I とその他の変数 v が同時に変化するとき, f の抽出を抑制すると変数 v が未定義になる. そのため, この様な未定義の変数を参照する文もプログラムスライシングにおける抽出を抑制する必要がある. 代入文 f において値が未定義である変数 $U(f)$, およびプログラムスライシングにおいて抽出しない文の集合 L としたとき, それらは以下のように計算される.

抽出しない文の集合 L**:** 代入文 f の直前が条件分岐である場合には直前に行われる代入は複数ありえるが, これらの集合を $prev(f)$ とする. 代入文 f が代入禁止実装変数群 Y_I を定義する場合, または f が未定義な変数群 $\bigcap_{g \in prev(f)} U(g)$ を参照する場合は f を L に追加する. ただし, f が変数群 X_I に対する代入である場合は追加しない. これにより, 細分化モデルで変化する変数が細分化実装においても変化する.

未定義変数群 $U(f)$**:** $U(f)$ は $\bigcap_{g \in prev(f)} U(g)$ を引き継ぐ. これにより, 条件分岐のいずれかによって値が定まるならば未定義変数群から取り除かれる. f が L に含まれるならば, f で定義される変数群を $U(f)$ に加える. そうでないならば f で定義される変数群を $U(f)$ から取り除く.

プログラムスライシングの適用

本稿ではプログラムスライシングとして 2.4.2 項で紹介した Weiser の手法を用いる. ただし，先の ‘抽出しない文の特定’ で示したように本手法では代入禁止大域変数群を考慮して Weiser の手法を適用する必要がある．そのため, Weiser の手法に対して抽出しない文の集合 L に含まれる文は抽出される文 S_C に加えない’ という拡張をする. Weiser の手法は入力として代入文 G, プログラム中の行 i，注目する変数群 V を入力とする．本稿では G を先の ‘制御構造の抽出’ で得られた代入文とし，i を実装操作の末尾，V を細分化モデルで変化する変数群のリンク変数とする．これにより細分化実装の操作が得られる.

B Method の実装は一般的な命令型言語と異なり WHILE ループに不変条件を与える. 本手法では不変条件を考慮せずに WHILE ループに Weiser の手法を適用する．WHILE ループの不変条件は後に行う証明責務最小化で実装の不変条件と同様に得る．この様に WHILE ループの不変条件をループ中の代入文とは独立して抽出するのは，WHILE ループの不変条件はループ中の代入文には影響を与えず，ループの停止性保証とループで生

成される値の特定に用いられるためである.

7.2.5　副作用解消

副作用解消は入力された代入文 V に対して V の参照部 V_{ref} と V の代入部 V_{def} を出力する. ここで, V の代入結果と V_{ref}; V_{def} の代入結果は等しい. 4.2.3 項の細分化実装の定義で述べたように, 細分化実装の代入は副作用を解消するために参照部と代入部に分割される. 副作用解消手順を下記の箇条書きに示す. 与えられた代入を参照部と代入部に分割するためには大域変数への代入文での大域変数の参照を回避しなければならない. そのため, 手順 (2) のように大域変数 v に格納される値 E を局所変数 X に代入し, v には X を代入する. ここで, 局所変数 X の計算が参照部であり, X を大域変数 v に代入するのが代入部である. モジュールの操作呼び出しにより大域変数が変化する場合には大域変数に代入される値を直接知ることが出来ないため, 手順 (3) の様に参照部で操作呼び出しのパラメータを局所変数群 $X_1, \ldots, X_n$ に代入し, 代入部ではそれを用いてモジュールの操作を呼び出す. 参照部は入力された代入の制御構造をそのまま維持するが, この制御構造において大域変数 v への代入が生じる条件 p と生じない条件 $\neg p$ がある場合, 条件 $\neg p$ に対して参照部は局所変数 X に値を代入しないため, 代入部において X の値が未定義になる. そのため, 手順 (1) のように代入部に参照部と同様の制御構造を作ることで，未定義な局所変数の参照を回避する.

1. IF 文 f の条件式それぞれに対して新たな局所変数群を宣言し, これに条件式の値を代入する. この局所変数群を参照することで条件分岐を代入部に複製する. すなわち, 条件式が p で, THEN 部 V_t, ELSE 部 V_e である IF 文 f に対して局所変数 $X_p := p$ を参照部で定義し, 条件式が X_p, THEN 部と ELSE 部が空である IF 文 f' を作る. ただし, IF 文 f の THEN 部 V_t, ELSE 部 V_e が IF 文を持つ場合, それらに対して再帰的に IF 文の複製を行い, 複製された IF 文を f' の THEN 部, ELSE 部に格納する.
2. 大域変数 v へ代入を行う文が条件式 p の制御ブロックに格納された等価代入文 $v := E$ であるならば, 新たな局所変数 (ここでは X とする) を宣言し, 大域変数への代入文を $X := E$ に置き換える. 置き換えて得られた操作を参照部 V_{ref} とする. また, 代入部 V_{def} の条件式 p の制御ブロックに代入文 $v := X$ を格納する. 代入文 $v := X$ は大域変数を参照しないため, 制御ブロック内であれば他の大域変数への代入文との代入順序は自由である.
3. 条件式 p の制御ブロックでモジュールの操作 $v.op(arg_1, \ldots, arg_n)$ を呼び出すことで大域変数への代入を行う場合, 新たな局所変数 (ここでは $X_1, \ldots, X_n$ とする) を宣言し, 操作呼び出しを $X_1 := arg_1; \ \ldots; \ X_n := arg_n$ に置き換える. 置き換えて得られた操作を参照部とする. また, 代入部の条件式 p の制御ブロックに $v.op(X_1, \ldots, X_n)$ を

格納する.

7.2.6　証明責務最小化

証明責務最小化はモデル, 実装, 細分化モデル, 及び, それらから 7.2.5 項の副作用解消で得られた操作 $V_I' = V_{ref}';\ V_{def}'$ を入力として実装依存モデルの制約 C_D', P_D', I_D', Q_D' と細分化実装の制約 P_I', I_I' を出力する. 出力において C_D', P_D', I_D', Q_D' は細分化モデルの制約 C_A', P_A', I_A', Q_A' に実装依存の制約を論理積で追加した物であり, 実装依存モデルと細分化実装から得られる式 (2.6) の証明責務は真になる. このような出力を得るため, 証明責務最小化では入力を証明責務に変換した上で, その証明責務が真であることを保つように証明責務を最小化する. これを実装依存モデルと細分化実装における証明責務とし, 証明責務から実装依存モデルと細分化実装を生成することで, 出力において証明責務が真になることを保証する. 証明責務の最小化は式 (4.7) の細分化実装の証明責務のゴールを定め, その証明責務が真になるように仮定を削減することで行う. 証明責務が真になるように仮定を削減するとは，ゴールが常に偽な命題を持つ場合には仮定が偽になるように背反する命題を仮定に残す．ゴールが常に偽な命題を持たないならば，ゴールを真にするのに必要な命題を仮定に残す．この様に仮定の削減にはゴールを定める必要があるため，ゴールにおいて未知である細分化実装の不変条件を仮に I_I'' と定め, 細分化実装の証明責務のゴールと元の実装の証明責務の仮定を持った式 (7.6) のような命題をつくり, その命題において命題の証明に寄与しない仮定を削減することで実装依存モデルと細分化実装における証明責務の要素を得る.

$$(C_A \wedge P_A \wedge I_A \wedge Q_A) \wedge (P_I \wedge I_I) \Rightarrow [V_I']\neg[V_A']\neg(I_I'' \wedge u = u') \tag{7.6}$$

式 (7.6) において u はモデル操作における戻値，u' は実装操作における戻値である．証明責務最小化手順を以下に示す.

1. 細分化実装の操作 V_I' に現れる変数, 細分化モデルに現れる変数, および, プリミティブな値や型のみで構成された実装の不変条件を仮の不変条件 I_I'' とする.
2. 式 (4.7) に対して仮定削減を行う. 仮定削減後の仮定が含む不変条件を I' とする.
3. I_I'' を $I' \wedge I_I''$ で置き換え, I' が変化しなくなるまで手順 (2), 手順 (3) を繰り返す.
4. 収束した仮定削減結果の仮定の C_A, P_A, I_A, Q_A と細分化モデルの制約の論理積から実装依存モデルのパラメタ制約 C_D', プロパティ制約 P_D', 不変条件 I_D', 事前条件 Q_D' を得る. また, 同様に仮定削減結果の P_I から細分化実装のプロパティ制約 P_I', I_I'' から不変条件 I_I' を得る.

上記手順において, 仮定削減は定理証明の過程で証明に寄与しない仮定を削減することで行う. そのため, 仮定削減は機械証明の補助を受けて人手によって行われる. この定理

証明で対象とする命題は細分化元ソフトウェアにおける証明責務に似た命題であり, 細分化元ソフトウェアで証明を行った作業者であれば容易に証明することが出来る.

7.3　部品登録

部品登録では MSFC とその仕様である細分化モデルを入力として ‘細分化モデルによる階層構造の構築’ と ‘モデル実装間変数対応付け’ を行うことで部品リポジトリを構築する.

7.3.1　階層構造の構築

4.4.2 項で述べたように本研究で扱う部品リポジトリは以下の性質を持つ．部品登録ではこれらの性質を満たすように部品をリポジトリに格納しなければならない.

1. 部品は細分化モデル毎に保存される.
2. 細分化モデルは階層構造をなす.

4.4.2 項で述べたように部品リポジトリの階層構造は ‘$M_1 \subseteq_S M_2$ ならば M_1 は M_2 の親である’ と定義される．この $M_1 \subseteq_S M_2$ は M_2 を検索キーとした部品検索において M_1 がマッチすることを表す．そのため，性質 (1) と性質 (2) を満たすために 6.2 節で定義した部品検索を応用する.

追加する部品の細分化モデルを C，階層構造の根となる細分化モデルを M_R としたとき，この階層構造への部品登録 $add(M_R, C)$ を以下のように定義する.

1. $M_R \subseteq_S C$ であり，M_R の子 M_a に対して $C \subseteq_S M_a$ であるならば，M_R と M_a の親子関係を解消し，M_a の親を C に，C の親を M_R にする.
2. M_R の子 M_a に対して $M_a \subseteq_S C$ であるならば，$add(M_a, C)$ を再帰的に行う.
3. 細分化モデルの階層構造が決定したら，その細分化モデルをインデックスとして持つ部品群の変数を 6.2.3 項の ‘変数名統一’ で細分化モデル C の変数名に統一する.
4. 7.3.2 項の部品の重複判定に従い，部品を登録する.

例えば，図 7.3 のように細分化モデル A, B, X, Y が格納されたリポジトリに部品を追加することを考える．この図において，a & b の様に細分化モデルに添えられた式は事前条件を表す．細分化モデル C_1 が $A \subseteq_S C_1$，$C_1 \subseteq_S X$ という関係を持つとき，C_1 を仕様として持つ部品をこのリポジトリに登録すると図 7.3(a) のように C_1 は A と X の親子関係に割り込む．C_1 に $B \subseteq_S C_2$ となる制約を加えた細分化モデル C_2 を持つ部品は図 7.3(b) のように A の子になり，A の代りに B, X の親になる.

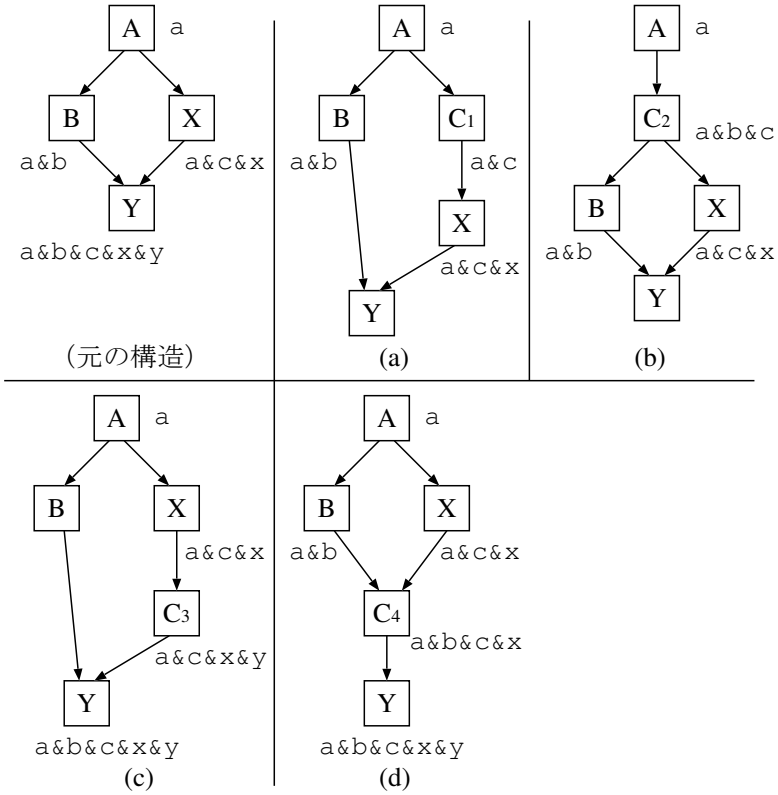


図 7.3: 部品登録時のリポジトリの構造変化

細分化モデル C_3, C_4 が $X \subseteq C_3$, $X \subseteq C_4$ である場合には $add(X, C_3)$，$add(X, C_4)$ を再帰的に行う．$C_3 \subseteq Y$ $C_4 \subseteq Y$ であるため，C_3, C_4 は図 7.3(c) 図 7.3(d) のように Y の親になる．

7.3.2　部品の重複判定

リポジトリに階層構造を構築する際に既に同じ細分化モデルを持つ部品が存在しても，本手法では部品をリポジトリに登録する．これは細分化モデルには実装依存の記述が存在せず，同じ細分化モデルを持つ部品でも情報の格納先などが異なるためである．しかし，一方で情報の格納先なども等しく，登録することに意味のない部品も存在しうる．このため，提案手法では以下のように部品の重複判定を行う．

1. 登録する部品の実装依存モデルが部品群の実装依存モデルと字面一致するかを判定する．いずれの実装依存モデルとも一致しないならば部品を登録する．
2. 実装依存モデルが一致するが，各モデル変数のリンク変数に対して輸入されるモジュールが異なる場合には部品を登録する．
3. 上記のいずれにも適合しない場合，部品は登録しない．

提案手法では 6.3.3 項の部品選択で述べたように各リンク変数で輸入するモジュールにより細分化モデルでは判定できない機能を選択する．逆説的にリンク変数に対するモジュールが等しいならば，部品選択において利用者は同一の機能を提供する部品と扱う．このため，部品の重複判定では利用者に判断を委ねずに部品の重複判定を行う．

このため，提案手法では‘計算結果は等しいがアルゴリズムの違いにより効率の異なる部品’ を保持，選択できない．この問題に対して本研究では‘計算結果が等しく，また，モジュールの判別により I/O の区別は可能であるため問題ない’ という立場を取っている．この事は効率等は計算の最適化などの研究により実現されるものであり，人間は計算結果に関係のない些末なアルゴリズムは考慮しなくて良いという考え方に基づく．

7.3.3　モデル実装間変数対応付け

4.4.3 項で述べたように，変数名置換を容易にするために各部品にはモデル変数 x_A とその変数に対応する実装変数 x_I の組 (x_A, x_I) を付加する．このモデル実装間変数対応は細分化実装のリンク不変条件から導出される．7.2.2 項で定義したようにリンク不変条件は実装の不変条件のうち, ‘モデル変数 = (実装変数の式)’ という命題であり, モデル変数と実装変数間の関係を表す．この実装変数の式が含む実装変数群がモデル変数に対応づけられる実装変数の候補となる．候補となる実装変数が 1 つの場合, モデル変数と実装変数が 1 対 1 に対応づけられる．この時, 他の候補から対応づけた実装変数を取り除くことで

候補の絞り込みを行う. 例えば, $A \subseteq B \subseteq C$ であるモデル変数 A, B, C に対して実装変数 x, y, z のリンク不変条件が $A = x$, $B = x \cup y$, $C = x \cup y \cup z$ であるときの変数対応付けを例示する. まず, 実装変数の候補が 1 つであるモデル変数 A について対応付け (A, x) が確定し, B, C の候補から x を取り除くことで (B, y) が, さらに C の候補から y を取り除くことで (C, z) が確定する. 経験的にモデルと実装の大域変数の数が同数ならば多くの場合に上記の絞り込みでモデル実装間変数対応付けを一意に定められる. 対応付けに失敗した部品は再利用できないためリポジトリから除外する.

モデル実装間変数対応付けの組み合わせは細分化モデルの変数の数に対して爆発するが, 細分化元のモデルが大きくなっても代入文が複雑化しない限り爆発しない. これは, 細分化モデルの粒度が '1 変数を変化させる代入' であり, 細分化モデルの変数の数が 1 代入文が含む変数と制御構造の制御変数に依存するためである. また, 同じソフトウェアから生成された部品は同じモデル実装間変数対応付けを持つため, 代入文が複雑化しても他の単純な細分化モデルで計算する事で計算量爆発を回避できる.

第8章

手法適用例

8.1 例題の概要

6 章と 7 章ではそれぞれモデル充足ソフトウェア合成 (MSSS) とモデル充足細粒度部品生成 (MSFC 生成) を説明した．6 章の MSSS の手順の説明では簡単な部品群を予め与えたが，3.3.3 項で述べたように部品リポジトリは MSFC 生成により整備される．本章ではこれを実演するために銀行口座システムに MSFC 生成を適用して部品リポジトリを整備する．さらに，銀行口座システムとグループ管理システムから得られた部品群が格納された部品リポジトリを用いて，マイレージ付きレンタカーシステムを MSSS で生成する．以下の節では本章で扱う各システムの説明を行う．

8.1.1 銀行口座システム

銀行口座システムのモデルの抜粋を図 8.1 に示す． 銀行口座システムは情報として顧客 ID(user)，氏名 (userName)，住所 (userAddr)，預金残高 (balance) を持ち，操作として顧客登録 (registUser)，預金預入 (deposit)，預金引出 (withdraw) を持つ．このシステムはファイルを用いて情報を保持するため，ファイルが開いているか否かを表すブール値 fileOpen を持つ．各操作の事前条件には `fileOpen=TRUE` という制約があるが，これは各操作の実行時にはファイルが開いている事を表す．

氏名，住所，預金残高は顧客 ID に対して一意に定まる．氏名と住所で用いる文字列は `seq(0..255)` の部分集合で表す．ここでは 0 から 255 の数値が 1 文字を表し，その順序列 (seq) によって文字列としている．`!(u1, u2)` から始まる不変条件において `!` は $\forall$ を表す．すなわち，‘あらゆる顧客間で氏名と住所の両方が等しい事はない’ という条件である．これは ‘氏名と住所の両方が等しい個人は同一人物である’，‘同一人物が二重登録されてはならない’ という要求を表している．

顧客登録操作 (registUser) は与えられた氏名と住所に対して未登録の顧客 ID を発行，登録し，顧客 ID を返す．与えられた氏名と住所が既に登録済である場合は登録を行わず，

```
MACHINE  Bank(maxUser, maxBalance)

/*パラメタ制約*/
CONSTRAINTS
  maxUser : 1..10000 & /*顧客ID上限*/
  maxBalance : 1..10000000 /*預金残高上限*/

/*不変条件*/
INVARIANT
  users <: 0..maxUser & /*顧客ID*/
  userName : users --> seq(0..255) & /*氏名*/
  userAddr : users --> seq(0..255) & /*住所*/
  balance : users --> 0..maxBalance & /*預金残高*/

  !(u1, u2).(u1:users & u2:users & not(u1 = u2) =>
   not(userName(u1)=userName(u2) & userAddr(u1)=userAddr(u2)))&
  [] /: ran(userName) &
  [] /: ran(userAddr) &
  fileOpen : BOOL

OPERATIONS
  /*顧客登録*/
  uid <-- registUser(name, addr) =
  PRE
    max(users) <= maxUser - 1 &
    name : seq(0..255) & addr : seq(0..255) &
    name /= [] & addr /= [] &
    fileOpen = TRUE
  THEN
    IF
      !(user).(user : users =>
        not(userName(user)=name & userAddr(user)=addr))
    THEN
      ANY newUser
      WHERE
        newUser : 0..maxUser &
        newUser /: users
      THEN
        users := users \/ {newUser} ||
        userName := userName <+ {newUser |-> name} ||
        userAddr := userAddr <+ {newUser |-> addr} ||
        balance := balance <+ {newUser |-> 0} ||
        uid := newUser
      END
    ELSE
      uid := -1
    END
  END;

  /*預金預入*/
  deposit(user, amount) =
  PRE
    user : users &
    amount : NAT &
    balance(user) + amount <= maxBalance &
    fileOpen = TRUE
  THEN
    balance(user) := balance(user) + amount
  END;

  /*預金引出*/
  withdraw(user, amount) =
  PRE
    user : users &
    amount : NAT &
    balance(user) - amount >= 0 &
    fileOpen = TRUE
  THEN
    balance(user) := balance(user) - amount
  END;
```

図 8.1: 銀行口座システムのモデル (抜粋)

```
輸入
  users_i.Set(0..maxUser),
  userName_i.StrPFun(0..maxUser),
  userAddr_i.StrPFun(0..maxUser),
  balance_i.PFunc((0..maxUser), (0..maxBalance)),
  CommonFunc

不変条件
  users = users_i.val &
  userName = userName_i.value &
  userAddr = userAddr_i.value &
  balance = balance_i.value &
  ((fileOpen = TRUE) => (users_i.fileOpen = TRUE)) &
  ((fileOpen = TRUE) => (userName_i.fileOpen = TRUE)) &
  ((fileOpen = TRUE) => (userAddr_i.fileOpen = TRUE)) &
  ((fileOpen = TRUE) => (balance_i.fileOpen = TRUE))

操作
 VAR uset, uu, uuName, uuAddr, corFlag, ii IN
  uset <-- users_i.getElements;
  ii <-- sizeOfSet(uset);
  corFlag := FALSE;
  WHILE ii > 0
  DO
    uu, uset <-- iterSet(uset);
    uuName <-- userName_i.getVal(uu);
    uuAddr <-- userAddr_i.getVal(uu);
    IF uuName = name & uuAddr = addr
    THEN corFlag := TRUE
    END;
    ii <-- sizeOfSet(uset)
  INVARIANT
    ii : NAT &
    ii = card(uset) &
    uset <: users_i.val &
    ((corFlag = FALSE) => !(user).(user : (users_i.val-uset) =>
     not(userName_i.value(user)=name & userAddr_i.value(user)=addr)))   (W1)
    ((corFlag = TRUE) => not(!(user).(user : (users_i.val-uset) =>
     not(userName_i.value(user)=name & userAddr_i.value(user)=addr))))  (W2)
  VARIANT ii
  END;


  IF corFlag = FALSE
  THEN
    VAR NID,isEmp, maxId IN
      isEmp <-- users_i.isEmpty;
      IF isEmp = TRUE
      THEN
        NID := 0
      ELSE
        maxId <-- users_i.getMax;
        NID := maxId + 1
      END;
      users_i.add(NID);
      userName_i.setVal(NID, name);
      userAddr_i.setVal(NID, addr);
      balance_i.setVal(NID, 0);
      uid := NID
    END
  ELSE
    uid := -1
  END
 END;
```

図 8.2: 銀行口座システムの実装 (抜粋)

```
MACHINE
  GroupManager(maxPerson, maxGroup, limit)

/*パラメタ制約*/
CONSTRAINTS
  maxPerson : 1..10000 & /*個人IDの上限*/
  maxGroup : 1..10000 /*グループIDの上限*/
  limit : NAT1 /*所属人数の上限*/

/*不変条件*/
INVARIANT
  persons <: 0..maxPerson & /*個人ID*/
  groups <: 0..maxGroup & /*グループID*/
  belong : persons +-> groups & /*所属*/
  !(group).(group:ran(belong) =>
    card(belong~[{group}]) <= limit) &
  fileOpen : BOOL

  /*グループ脱退*/
  leaveGroup(user) =
  PRE
    user : dom(belong) &
    fileOpen = TRUE
  THEN
    belong := {user} <<| belong
  END;

  /*IDの発行，登録*/
  newId <-- addPerson(group) =
  PRE
    group : 0..maxGroup &
    group : groups &
    max(persons) <= maxPerson-1 &
    card(belong~[{group}]) + 1 <= limit &
    fileOpen = TRUE
  THEN
    ANY
      person
    WHERE
      person : 0..maxPerson &
      person /: persons
    THEN
      persons := persons ∨ {person} ||
      belong := belong <+ {person |-> group} ||
      newId := person
    END
  END;
```

図 8.3: グループ管理システムのモデル (抜粋)

顧客 ID の代りに-1 を返す．顧客登録操作では未登録の顧客 ID が発行できることを保証するために，事前条件で登録済の顧客 ID の最大値が顧客 ID の上限に達していないことを保証している．預金預入操作 (deposit) は与えられた金額を預金残高に加算する．預金引出操作 (withdraw) は与えられた金額を預金残高から減算する．預金預入と引出では事前条件によって加算と減算の結果が口座残高の上限を越えないことを保証している．

8.1.2　グループ管理システム

グループ管理システムのモデルの抜粋を図 8.3 に示す．グループ管理システムは説明会場などで来客者などに個人を識別する番号札などを配布し，来客者が参加する説明ツアーを管理する場合などに用いるシステムである．このシステムでは個人は ID 発行時に所属するグループが与えられるが，後でグループから脱退したりグループを変更できる．

このシステムは情報として個人 ID(persons)，グループ ID(groups)，所属 (belong) を持ち，操作として ID の発行登録 (addPerson)，グループ脱退 (leaveGroup) を持つ．このシステムは銀行口座システムと同様にファイルを用いて情報を保持するため，ファイルが開いているか否かを表すブール値 fileOpen を持つ．各操作の事前条件には `fileOpen=TRUE` という制約があるが，これは各操作の実行時にはファイルが開いている事を表す．

不変条件の所属についての制約に `persons +-> groups` という式があるが，これは個人 ID からグループ ID への部分関数を表す．一般的に関数は定義域に与えられた集合全体を網羅しなければならないが，部分関数は与えられた集合の部分集合を定義域とする．これによりグループに所属しない個人が存在しうる．また，`!(group)` から始まる所属に対する不変条件は‘各グループの所属人数は limit 以下’という制約である．操作 addPerson は事前条件として `card` から始まる制約をもつが，この条件は‘与えられたグループ (group)

```
MACHINE
  RentalCar(maxCar, maxUser, maxPoint, limit)
/*パラメタ制約*/
CONSTRAINTS
  maxPoint : 1..10000000 &
  maxCar   : 1..10000 &
  maxUser  : 1..10000 &
  limit : NAT1

/*不変条件*/
INVARIANT
  cars <: 0..maxCar & /*車両*/
  users <: 0..maxUser & /*顧客*/
  points : users --> 0..maxPoint & /*マイレージ*/
  userName : users --> seq(0..255) & /*氏名*/
  userAddr : users --> seq(0..255) & /*住所*/
  rent : cars +-> users & /*貸出*/
  !(user).(user : users => card(rent~[{user}]) <= limit) &
  !(u1, u2).(u1:users & u2:users & not(u1 = u2) =>
   not(userName(u1)=userName(u2) & userAddr(u1)=userAddr(u2)))&
  [] /: ran(userName) &
  [] /: ran(userAddr)

/*顧客登録*/
  uid <-- addUser(name, addr) =
  PRE
    max(users) <= maxUser - 1 &
    name : seq(0..255) &
    addr : seq(0..255) &
    name /= [] &
    addr /= []
  THEN
    IF
      !(user).(user : users =>
       not(userName(user)=name & userAddr(user)=addr))
    THEN
      ANY newUser
      WHERE
        newUser : 0..maxUser &
        newUser /: users
      THEN
        users := users \/ {newUser} ||
        userName := userName <+ {newUser |-> name} ||
        userAddr := userAddr <+ {newUser |-> addr} ||
        points := points <+ {newUser |-> 0} ||
        uid := newUser
      END
    ELSE
      uid := -1
    END
  END;

/*車両貸出*/
  rentCar(user, car) =
  PRE
    user : users &
    car : cars &
    car /: dom(rent) &
    card(rent~[{user}]) + 1 <= limit
  THEN
    rent := rent <+ {car |-> user}
  END;


/*車両返却*/
  returnCar(car, user, point) =
  PRE
    car : cars &
    user : users &
    rent(car) = user &
    point : NAT &
    points(user) + point <= maxPoint
  THEN
    rent := {car} <<| rent ||
    points(user) := points(user) + point
  END
```

図 8.4: マイレージ付きレンタカーシステムの要求モデル

の所属人数が limit 未満’ を表す．これにより addPerson 実行後に所属人数が limit を越えないことを保証している．

8.1.3　マイレージ付きレンタカーシステム

マイレージ付きレンタカーシステムの要求モデルを図 8.4 に示す．先の銀行口座システムやグループ管理システムと異なり実装方法を与えていないためこのモデルは fileOpen などの実装依存の変数や制約を持たない．マイレージ付きレンタカーシステムは通常のレンタカー同様に顧客に対して車両を貸し出すが，顧客はマイレージというポイントを持ち，このポイントは車両の返却時に加算される．

このシステムは情報として車両 (cars)，顧客 (users)，マイレージ (points)，氏名 (userName)，住所 (userAddr)，貸出 (rent) を持ち，操作として顧客登録 (addUser)，車両貸出 (rentCar)，車両返却 (returnCar) を持つ．このシステムは銀行口座システムと同様に顧客は氏名と住所を持ち，氏名と住所の両方が重複する顧客は存在しない．また，`!(user)`

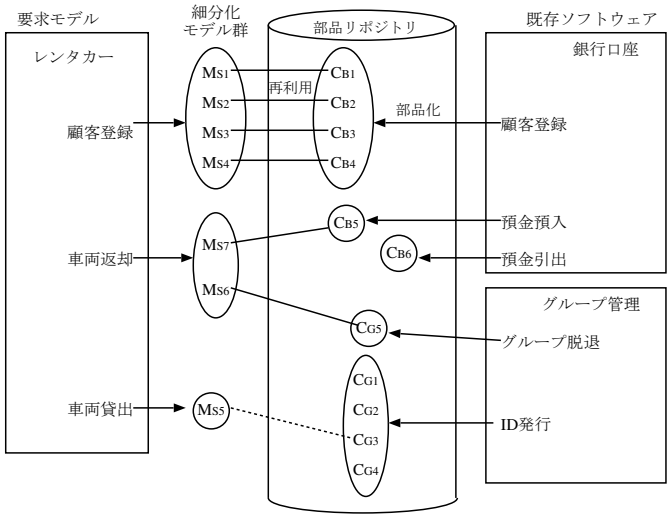


図 8.5: 要求モデルの操作と再利用される部品の生成元操作

から始まる不変条件によって各顧客が一度に借りられる車両の台数は limit に制限される．このため，車両貸出には事前条件として `card` から始まる制約が与えられている．この制約は ‘与えられた顧客 (user) が借りている車両の台数は limit 未満’ を表しており，これにより，貸出台数が 1 台増えても limit を越えないことが保証される．

8.2　手法適用の流れ

8.1 節に述べたように本章では銀行口座システムとグループ管理システムから部品群を生成し，それらを利用してマイレージ付きレンタカーシステムを構築する．

図 8.5 は要求モデルにモデル細分化を適用して得られる検索キー群 $M_{S1}, \ldots, M_{S7}$，銀行口座システムから得られる部品群 $C_{B1}, \ldots, C_{B6}$, グループ管理システムから得られる部品群 $C_{G1}, \ldots, C_{G5}$ の細分化元操作と再利用の関係を表したものである．図中で検索キー M_{S5} と部品 C_{G3} のみが点線で結ばれるが，これは部品検索時に M_{S5} に対して C_{G3} がマッチせず，‘不完全なソフトウェアへの対応’ で部品が再利用される事を表す．

銀行口座システムに MSFC 生成を適用すると初期化からは 2 つ，顧客登録からは 4 つ，預金預入からは 1 つ，預金引出からは 1 つの部品が生成される．グループ管理システムに MSFC 生成を適用すると初期化からは 3 つ，ID 発行からは 4 つ，グループ脱退からは 1 つの部品が生成される．これに対してレンタカーシステムに MSSS を適用すると初期化からは 4 つ，顧客登録からは 4 つ，車両貸出からは 1 つ，車両返却からは 2 つの検索キーが得られる．これらの検索キーに対して図 8.5 のように部品リポジトリに格納された銀行口座システムとグループ管理システムから得られた部品群を再利用することでレンタカーシステムに対する合成モデルと合成実装が得られる．

図 8.5 においてレンタカーシステムの顧客登録は銀行口座システムの顧客登録から生成された部品を全て再利用する事で実装される．車両返却は銀行口座システムの預金預入とグループ管理システムのグループ脱退から生成された部品をそれぞれ一つ再利用することで実装される．

8.3　MSFC 生成の適用例

ここでは銀行口座システムの顧客登録に対して MSFC 生成を適用する例を示す．図 8.5 で示したように顧客登録操作からは 4 つの部品が得られる．本例では 4 つの部品の内，特に細分化モデルが ANY 文を持つ例を扱う．

8.3.1　モデル細分化

本節では 8.1.1 項で紹介した銀行口座システムの顧客登録操作に対してモデル細分化を適用する． 顧客登録の操作に対して非決定的値生成の分離を適用すると図 8.6 の様に非決定的値生成操作と非決定的値参照操作が得られる．次に，不変条件と事前条件に制約条件展開を適用すると図 8.7 のように不変条件と事前条件が列挙される．次に非決定的操作分割で得られた操作に操作分割を適用することで図 8.6 太枠の様な代入群が得られる．これらの代入群をそれぞれ細分化モデルの代入群とし，図 8.7 の不変条件と事前条件に制約条件抽出を適用することで各細分化モデルの制約条件が決定する．以下の節では図 8.6 の非決定的値生成操作の代入を代入文として持つ細分化モデルを生成する手順を説明する．

非決定的値生成の分離

5.2 節で述べたように非決定的値生成の分離ではモデルの操作を入力として非決定的値生成操作と非決定的値参照操作を生成する．非決定的値生成操作は元の操作と同じ事前条件を持ち，ANY 文の THEN 部として非決定的値 `v` を戻値 `res_v` に代入する代入文を持つ．また，非決定的値生成操作では `res_v` 以外の値への代入を行わない．

図 8.6 の分割元操作から非決定的値生成操作を分離すると ANY 文の THEN 部で 非決定的値 `newUser` を戻値に代入する操作が得られる．また，それ以外の変数への代入文は削除されるため，SELECT 文の ELSE 部は空になる．非決定的値参照操作は ANY 文の非決定的値を操作引数とし元の操作の事前条件 (Q) に加えて ANY 文の WHERE 部 (W) を事前条件に持つ．また，元の ANY 文を ANY 文の TEHN 部 (T) で置き換えた代入文を持つ．

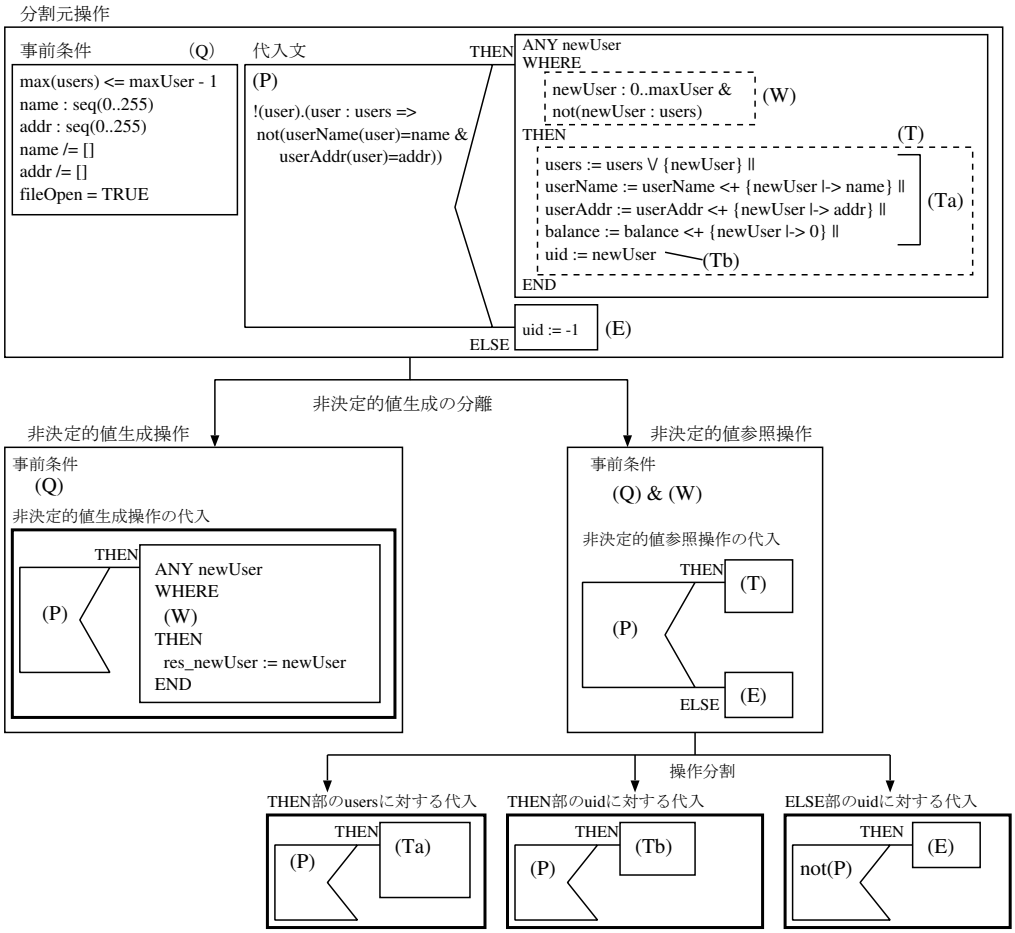


図 8.6: 顧客登録操作の非決定的値生成分離，操作分割結果

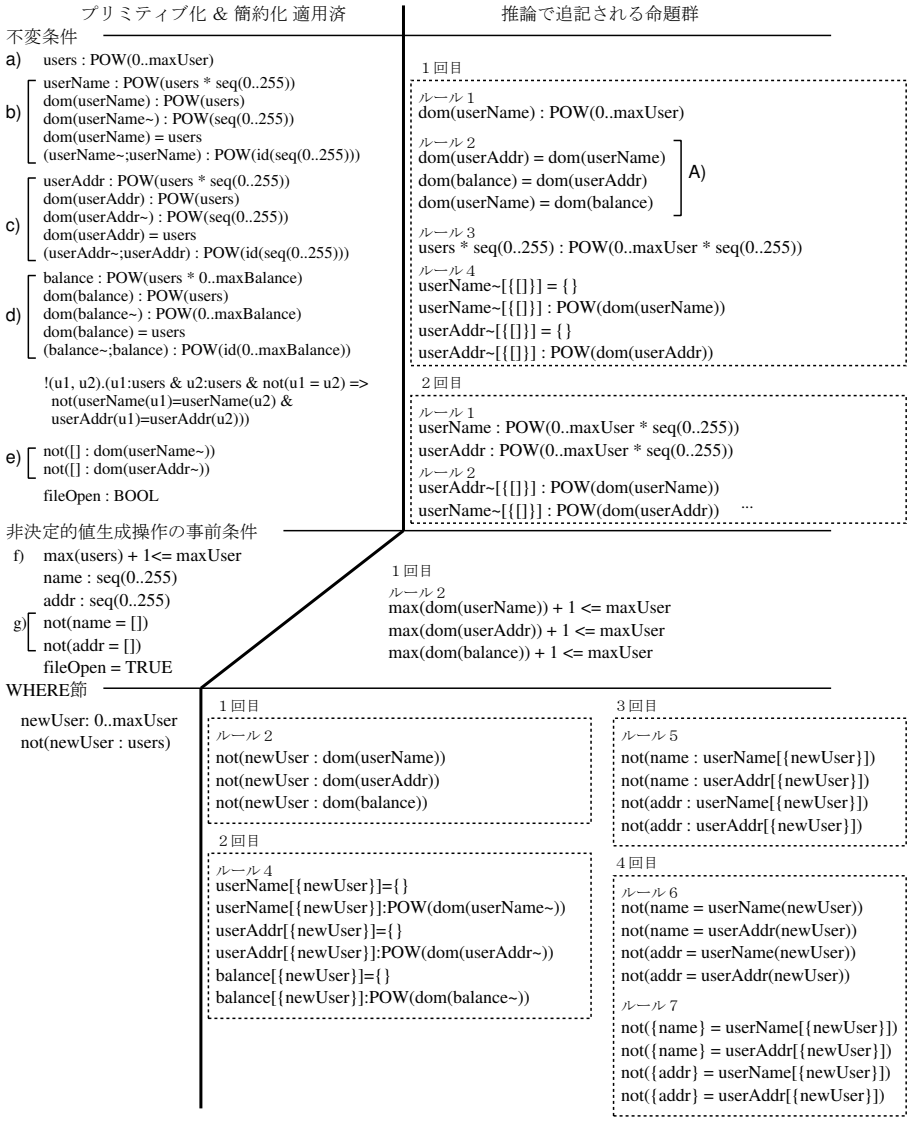


図 8.7: 顧客登録操作の制約条件展開結果

表 8.1: 銀行口座システムで適用されるプリミティブ化書き換え規則

番号	書き換え元	書き換え後
1	$a \subseteq b$	$a \in \mathbb{P}(b)$
2	$a \nrightarrow b$	$\{r \mid r \in a \leftrightarrow b \wedge \mathrm{dom}(r) \subseteq a \wedge \mathrm{ran}(r) \subseteq b \wedge (r^{-1};\ r) \subseteq \mathrm{id}(b)\}$
3	$a \rightarrow b$	$\{r \mid r \in a \nrightarrow b \wedge \mathrm{dom}(r) = a\}$
4	$a \leftrightarrow b$	$\mathbb{P}(a \times b)$
5	$\mathrm{ran}(r)$	$\mathrm{dom}(r^{-1})$
6	$a \notin b$	$\neg(a \in b)$
7	$a \neq b$	$\neg(a = b)$
8	$a \geqslant b$	$b \leqslant a$
9	$a \leqslant b - c$	$a + c \leqslant b$

表 8.2: 銀行口座システムで適用される推論規則

番号	規則
1	$X \in \mathbb{P}(Y) \wedge Y \in \mathbb{P}(Z) \Rightarrow X \in \mathbb{P}(Z)$
2	$(a = b),\ R(b, x) \Rightarrow R(a, x)$
3	$(X \times Y$ が記述中に在る$),\ X \in \mathbb{P}(Z) \Rightarrow X \times Y \in \mathbb{P}(Z \times Y)$
4	$r[\{x\}] = \emptyset \wedge r[\{x\}] \in \mathbb{P}(\mathrm{dom}(r^{-1})) \Leftrightarrow \neg(x \in \mathrm{dom}(r))$
5	$(S \in \mathbb{P}(Y),\ y \in Y),\ S = \emptyset \Rightarrow \neg(y \in S)$
6	$(r \in \mathbb{P}(a \times b),\ (r;\ r^{-1}) \in \mathbb{P}(b)),\ y \in r[\{x\}] \Leftrightarrow y = r(x)$
7	$(r \in \mathbb{P}(a \times b),\ (r;\ r^{-1}) \in \mathbb{P}(b)),\ y \in r[\{x\}] \Leftrightarrow \{y\} = r[\{x\}]$

制約条件展開

5.3 節で述べたように，制約条件展開はプリミティブ化，簡約化，推論による式の追記，主加法標準化の 4 段階からなる．図 8.7 の左側は図 8.6 の非決定的値生成操作の不変条件，値定義操作の事前条件 (Q)，ANY 文の WHERE 節 (W) にプリミティブ化と簡約化を適用した結果である．図 8.7 の左側は簡約化結果に推論による式の追記を行った結果である．この例では制約条件が論理和を含まないため主加法標準化による変化はない．

表 8.1 は本例のプリミティブ化で実際に適用された書き換え規則のみを抜粋した表である．また，表 8.2 は本例の推論による式の追記で実際に適用された推論ルールのみを抜粋した表である．以下ではプリミティブ化における書き換え規則適用と式の追記における推論ルール適用を説明する．

プリミティブ化は図 8.1 の不変条件，および，図 8.6 の事前条件 (Q), WHERE 節 (W) に対して適用する．図 8.7 の不変条件 a) は書き換えルール 1 (部分集合から巾集合への書き換え) により得られる．また，b), c), d) はルール 2, 3, 4, 5 (写像から集合への書き換

え) を再帰的に適用することで得られる．e), g) は ルール 6, 7 (否定的関係演算子の ¬ への書き換え) を適用することで得られる．また，f) は ルール 8, 9 (不等号の統一) を適用することで得られる．

次に図 8.7 左側の命題群に対して表 8.2 の推論規則を収束するまで適用し，制約条件を展開する．図 8.7 右側の 1 回目，2 回目という表記は何回繰り返したときに推論されたかを表し，ルール番号は表 8.2 のどの規則により推論されたかを表す．この例では図のように不変条件に対しては 2 回，事前条件に対しては 1 回，WHERE 節に対しては 4 回推論を繰り返すことで推論結果が収束した．特に重要な推論として，ここでは WHERE 節に対する推論ルール適用を説明する．WHERE 節の命題 `not(newUser:users)` に対して `users = dom(userName)` であるため，ルール 2 が適用され，`users` を `dom(userName)` に置換した命題が追加される．同様の推論は `userAddr`，`balance` に対しても適用される．2 回目の推論では 1 回目の推論結果に対してルール 4(右から左へ) を適用することで，定義域に含まれない値 newUser に対する写像が空であるという命題が得られる．3 回目の推論では ルール 5 の S を `userName[{newUser}]`，Y を `dom(userName~)`，y を `name` と置くことで newUser に対する写像結果が name を含まないという命題が得られる．4 回目の推論は ルール 6, 7 の逆を適用することで得られる．

操作分割

図 8.6 の非決定的値定義操作と非決定的値参照操作に対して 5.4 節の操作分割を適用する．操作分割ではまず等価な変数群を特定する．制約条件展開で得られた命題群 (図 8.7) において，A) の命題群から {users, userName, userAddr, balance} の 4 変数が 1 つの等価な変数と判定される．このため，これらの変数に対する代入は分割されず，図 8.6 の ‘THEN 部の users に対する代入’ は {users, userName, userAddr, balance} に対する同時代入 (T_a) となる．

図 8.6 の非決定的値生成操作に対する操作分割結果は非決定的値生成操作そのものになる．非決定的値生成操作の ‘注目する変数’ が res_newUser ただ一つであり，また，ANY 文の THEN 部も res_newUser への代入のみで構成されるためである．

図 8.6 の非決定的値参照操作に対する操作分割結果は図 8.6 のように ‘THEN 部の users に対する代入’，‘THEN 部の uid に対する代入’，‘ELSE 部の uid に対する代入’ となる．まず，非決定的値操作の代入は条件式 (P) に対して THEN 部 (T) と ELSE 部 (E) で構成される．ここで，THEN 部と ELSE 部の条件はそれぞれ，(P) と ¬(P) であり，条件節が排他的であることから，(T) と (E) が分割される．さらに，(T) を分割すると，{users, userName, userAddr, balance} を注目する変数とした代入文 (T_a) と {uid} を注目する変数とした代入文 (T_b) に分割される．同様に (E) では注目する変数が {uid} のみであり，(E) の分割結果は (E) そのものになる．以上により，‘SELECT (P) THEN (T_a) END’,

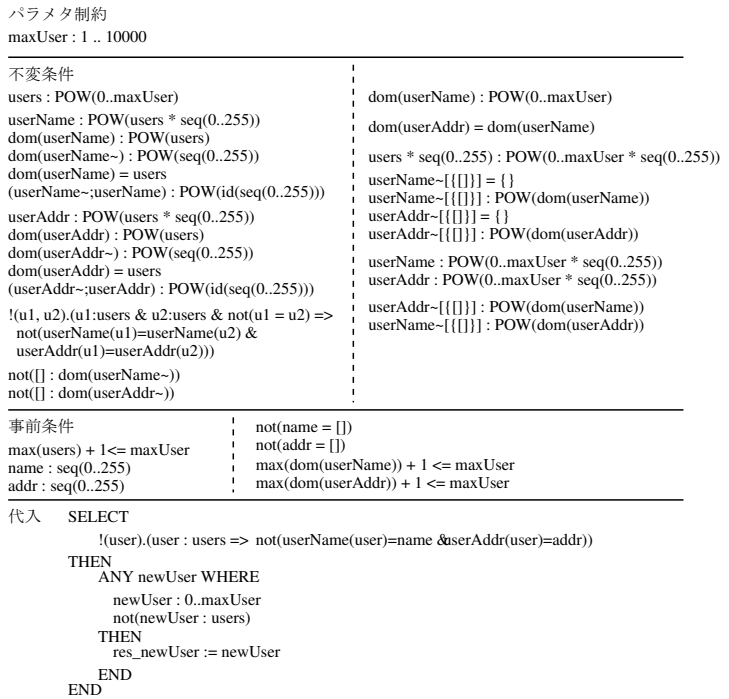


図 8.8: 図 8.6 の非決定的値生成操作についての制約条件抽出結果

‘SELECT (P) THEN (T_B) END’, ‘SELECT $\neg(P)$ THEN (E) END’ が非決定的値参照操作への操作分割結果として得られる．

制約条件抽出

展開済の制約条件 (図 8.7) に対して操作分割で得られた 4 つの操作群それぞれについて制約条件抽出 (5.5 節) を適用する．ここでは，5.5 節に沿って図 8.6 の非決定的値生成操作についての制約条件抽出を適用し，図 8.8 の制約条件を得る過程を例示する．

まず，5.5 節の記法に則り，細分化前の代入文を V，非決定的値生成操作から操作分割で得られた代入文を V'，V' で変化する変数群を X，V で変化するが，V' で変化しない変数群を X' とする．このとき，制約条件抽出に関連する変数群は以下のようになる．

vars(V'): users, userName, userAddr, name, addr, user, newUser, res_newUser

X: res_newUser

X': users, userName, userAddr, balance, uid

5.5 節の制約条件抽出により，細分化モデルの事前条件は図 8.7 の展開済事前条件のうち，vars(V') とプリミティブのみで記述された命題のみである．制約条件抽出ではモデル引数 (maxUser, maxBalance) は定数として扱われるため，この例では fileOpen と balance を

含まない事前条件が全て抽出される．また，細分化モデルの不変条件は図 8.7 の展開済不変条件のうち，‘(vars(V') - X) とプリミティブのみで記述された不変条件’ と ‘(vars(V') - X') とプリミティブのみで記述された不変条件’ の論理積である．この例では ‘(vars(V') - X') とプリミティブのみで記述された不変条件’ は存在しない．そのため，‘(vars(V') - X) とプリミティブのみで記述された不変条件’ のみを考えれば良い．この結果，fileOpen と balance を含まない不変条件が全て抽出される．

モデル引数 maxUser, maxBalance のうち，maxBalance は操作分割結果，抽出された不変条件と事前条件のいずれにも含まれない．よって，maxBalance はモデル引数から除外され，パラメタ制約も maxUser についての制約のみが残される．

構文要素整列

図 8.8 の制約条件と代入に対して構文要素整列を適用することで字面の統一された細分化モデルが得られる．本節では図 8.8 の制約条件と代入から一部を抜粋して構文要素整列手順を示す．

図 8.9 は図 8.8 から抜粋した制約条件と代入を構文木で表したものである．まず，変数の初期重み付けを適用する．事前条件以外の制約条件において，$a = b$, $a \leqslant b$, $a \in b$, $a \in \mathbb{P}(b)$ に該当する変数 a, b の組は (u1, users) と (u2, users) の 2 つである．u1, u2 は全称記号 ！ の式中で宣言される変数であり，構文木 (g) の部分木 (g1) と (g2) において u1 ∈ users, u2 ∈ users と定義される．これにより, u1 $>_w$ users, u2 $>_w$ users と重みが定まる．同様に構文木 (o) の部分木 (o1) から user $>_w$ users が定まる．

次に被演算式順序統一を適用する．図 8.9 において，可換演算子は点線で表現されている．例えば，部分木 (g1), (g2), (g3) を被演算式として持つ論理積 (&) では (g1) ∧ (g2) ∧ (g3) と (g2) ∧ (g1) ∧ (g3) の意味は等しい．被演算式順序統一は被演算式に対して再帰的に行う．この例では (g3) が可換演算子 = を持つため，その被演算式 u1, u2 に対して被演算式順序統一を適用してから，(g1), (g2), (g3) の順序を定める．変数 u1, u2 間の重みは未決定なため，(g3) の可換演算子 = の被演算式の順序は未決定なままである．さらに (g1), (g2), (g3) に式順序付けを適用すると，‘{(g1), (g2)}, (g3)’ という順序が定まる．ここで {} は同順を表す．この式順序付けは構文要素の重みの判定により行われる．表 8.3 は 5.6.4 項の表 5.3 の抜粋である．この表中で (g1), (g2), (g3) の根である演算子 ‘:’(∈) と ‘not’ (¬) は: $>_w$ not と定められる．よって，式順序 ‘{(g1), (g2)}, (g3)’ が定まる．また，論理積の被演算式である (o2), (o3) についても同様に ‘(o2), (o3)’ と順序が定まる．その他の被演算式は変数 u1 と u2 間，userName と userAddr 間，name と addr 間の重みが未定義であるため，決定されない．

次に，構文木 (a) から (o) に式順序付けを適用する．まず，式が属する節 (パラメタ制約, 不変条件，代入) により (a), {(b), (c), (d), (e), (f), (g), (h), (i)}, (o) が決定する．さらに，

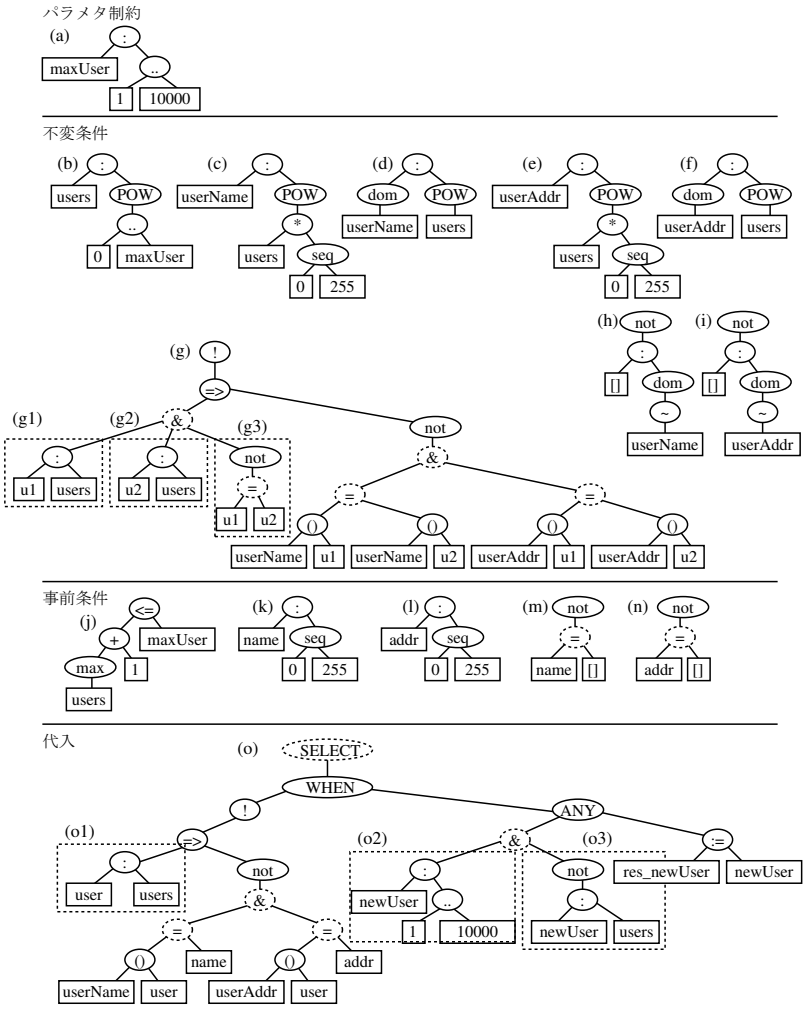


図 8.9: 図 8.8 から得られる構文木 (抜粋)

表 8.3: 構文要素の重み (抜粋)

W	要素	W	要素	W	要素	W	要素
20	変数	14	$+esc+le$	8	$+esc+upto$	2	$+esc+forall$
19	プリミティブな値	13	$+esc+neg$	7	[]	1	$+esc+max$
18	関数呼び出し	12	ANY	6	$+esc+dom$		
17	$+$	11	SELECT	5	$+esc+times$(直積)		
16	$+esc+in$	10	:=	4	$^{-1}$(逆像)		
15	$=$	9	$+esc+power$	3	seq		

不変条件の式 {(b), (c), (d), (e), (f), (g), (h), (i)} に対して重みによる式順序付けを適用すると，‘(b), {(c), (e)}, {(d), (f)}, {(h), (i)}, (g)’ と順序が定まる．(c) と (e) 間，(d) と (f) 間，(h) と (i) 間で順序が未決定であるのは演算子と構文木の構造が等しく，userName と userAddr 間で変数間の重みが未決定であるためである．構造の異なる構文木間で構文要素の重みを比較する．これらの根である演算子 ‘:’(∈)，‘not’(¬)，‘!’(∀) の重みを比較すると，表 8.3 より ‘:’ $>_w$ ‘not’ $>_w$ ‘!’ であるため，{(b), (c), (d), (e), (f)}, {(h), (i)}, (g) が定まる．次に，{(b), (c), (d), (e), (f)} において根の第 1 子の重みを比較すると ‘変数’ $>_w$ ‘dom’ より {(b), (c), (e)}, {(d), (f)} が定まる．{(b), (c), (e)} において根の第 2 子はいずれも ‘POW’(ℙ) であるため，さらにその子の重みを比較すると ‘..’ $>_w$ ‘∗’(直積) より，(b), {(c), (e)} が定まる．以上により式順序が ‘(a), (b), {(c), (e)}, {(d), (f)}, {(h), (i)}, (g), (o)’ と定まる．

次に，変数重み付けを適用する．先の式順序付けで定まった順序で構文木を幅優先探索し，出現順に変数を並べると maxUser, users, {userName, userAddr}, {u1, u2}, user, {name, addr} となる．この変数重みで再度，被演算式順序付けと式順序付けを適用する．しかし，この例では再度適用しても式順序が新たに定まらない．このため，5.6.1 項で示したように，userName と userAddr 間で適当な重み ‘userName $>_w$ userAddr’ を定めて整列を繰り返す．この結果，全ての可換演算子の被演算式順序が定まり，式順序 ‘(a), (b), (c), (e), (d), (f), (h), (i), (g), (o)’ が決定する．また，変数の重みも maxUser $>_w$ users $>_w$ userName $>_w$ userAddr $>_w$ u1 $>_w$ u2 $>_w$ user $>_w$ name $>_w$ addr と定まる．

ちなみに，ランダムに定めた重み ‘userName $>_w$ userAddr’ を逆，すなわち ‘userAddr $>_w$ userName’ としても整列結果は等しい．これは双方の整列結果の変数を登場順に x_1 から x_9 に置換すると字面が等しいことから明らかである．

最後に上記の変数重みを用いて事前条件に被演算式順序付けと式順序付けを適用し，(k), (l), (j), (m), (n) と式順序が決定する．

未整列の細分化モデル (図 8.8) に対する整列結果を図 8.10 に示す．

8.3.2　実装抽出

7.2 節で述べたように実装抽出はモデル，実装，細分化モデルを入力として部品，すなわち細分化実装と実装依存モデルを出力する．本例では 8.1.1 項で紹介した銀行口座システムのモデルと実装，および，前節で生成した細分化モデル (図 8.10) を入力として部品を生成する．図 8.11 に本例で得られる部品を示す．

以下の節では実装抽出を ‘大域変数の対応付け’，‘非決定的値生成の分離’，‘操作抽出’，‘副作用解消’，‘証明責務最小化’ と順を追って説明する．

パラメタ制約
maxUser : 1 .. 10000

不変条件
users : POW(0..maxUser)
userName : POW(users * seq(0..255))
userAddr : POW(users * seq(0..255))
userName : POW(0..maxUser * seq(0..255))
userAddr : POW(0..maxUser * seq(0..255))
userName~[{[]}] : POW(dom(userName))
userName~[{[]}] : POW(dom(userAddr))
userAddr~[{[]}] : POW(dom(userName))
userAddr~[{[]}] : POW(dom(userAddr))
dom(userName) : POW(users)
dom(userAddr) : POW(users)
dom(userName) : POW(0..maxUser)
dom(userAddr) : POW(0..maxUser)
dom(userName~) : POW(seq(0..255))
dom(userAddr~) : POW(seq(0..255))

(userName~;userName) : POW(id(seq(0..255)))
(userAddr~;userAddr) : POW(id(seq(0..255)))
users * seq(0..255) : POW(0..maxUser * seq(0..255))
userName~[{[]}] = {}
userAddr~[{[]}] = {}
dom(userName) = users
dom(userAddr) = users
dom(userName) = dom(userAddr)
not([] : dom(userName~))
not([] : dom(userAddr~))
!(u1, u2).(u1:users & u2:users & not(u1 = u2) =>
  userAddr(u1)=userAddr(u2) &
  not(userName(u1)=userName(u2)))

事前条件
name : seq(0..255)
addr : seq(0..255)
max(users) + 1<= maxUser

max(dom(userName)) + 1 <= maxUser
max(dom(userAddr)) + 1 <= maxUser
not(name = [])
not(addr = [])

代入　SELECT
    !(user).(user : users => not(name=userName(user) &ddr=userAddr(user)))
  THEN
    ANY newUser WHERE
      newUser : 0..maxUser
      not(newUser : users)
    THEN
      res_newUser := newUser
    END
END

図 8.10: 図 8.6 の非決定的値生成操作についての細分化モデル

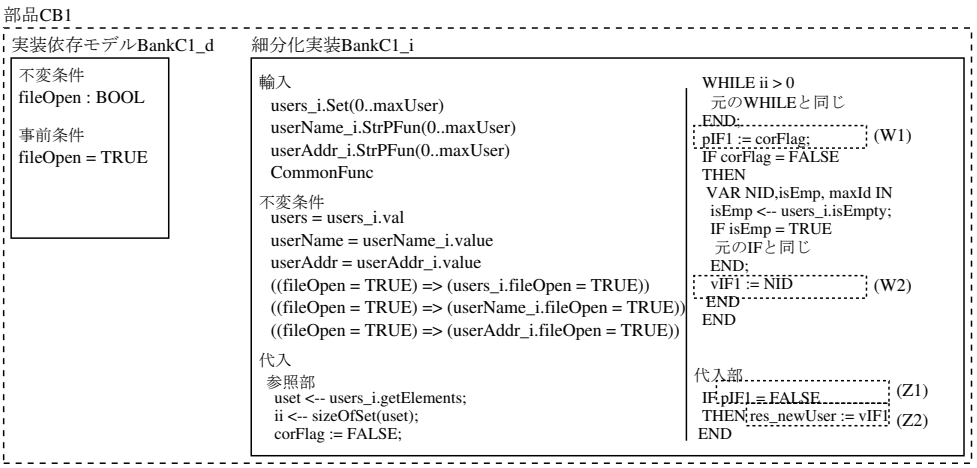


図 8.11: 図 8.10 を仕様とする部品例

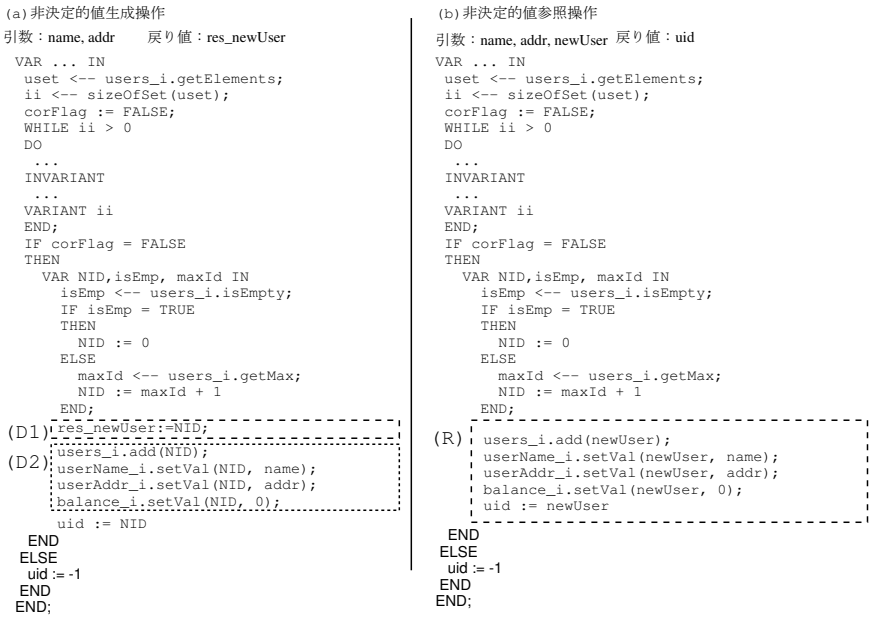


図 8.12: 銀行口座システム顧客登録の実装における非決定的値生成，参照操作

大域変数の対応付け

7.2.2 項の大域変数の対応付けではモデルの大域変数 x に対応するリンク変数群 $y_1, \ldots, y_k$ を出力する．図 8.2 の不変条件においてリンク不変条件は $x = E(Y)$ で表される．BankC1 のモデル変数 user, userName, userAddr についてのリンク不変条件はそれぞれ users = users_i.val, userName = users_i.value, userAddr = users_i.value と定義される．これらのリンク不変条件から users，userName, userAddr に対応するリンク変数群 user_i.val, userName_i.value, userAddr_i.value が決定する．

非決定的値生成の分離

7.2.3 項で説明したように非決定的値生成の分離では非決定的値 v に対応する式 E を特定する．さらに，式 E を決定し，それを戻り値として返す非決定的値生成操作と式 E の値を引数として受け取り，それを用いて代入を行う非決定的値参照操作を構築する．

図 8.1 の顧客登録において newUser が非決定的値 v である．非決定的値に対応する式 E は newUser を用いたモデルの代入文 `users:=users\/{newUser}` に対応する実装の代入文を特定することで行う．先の大域変数の対応付けより users に対応するリンク変数は users_i.val である．モジュール Set の操作 `add(xx)` は `val:=val\/{xx}`と定義されるため，このモジュール Set の操作 add を呼び出す `users_i.add(NID)` が顧客登録の実装操作において users_i.val に対する唯一の代入文となる．これにより，非決定的値 newUser に対応する式は局所変数 NID であると特定される．

```
((corFlag = FALSE) => !(user).(user : (users - uset) =>
   not(userName(user)=name & userAddr(user)=addr)))        (W1)
((corFlag = TRUE) => not(!(user).(user : (users-uset) =>
   not(userName(user)=name & userAddr(user)=addr))))       (W2)
```

図 8.13: 図 8.2 の WHILE 不変条件のモデル変数による表現

非決定的値生成操作は戻り値として res_newUser を持ち，非決定的値 newUser に対応する式 NID を含む大域変数への代入文の直前に `res_newUser:=NID` を追加した操作である．これにより非決定的値生成操作は図 8.12(a) のように元の操作に (D1) の一行を追加した操作となる．また，非決定的値参照操作は引数として newUser を持ち，大域変数への代入文において非決定的値に対応する式 NID を引数 newUser に置換した操作である．これにより，非決定的値参照操作は図 8.12(b) の代入 (R) のように NID が newUser に置換された操作である．

操作抽出

操作抽出は 7.2.4 項に示したように ‘制御構造の抽出’, ‘抽出しない文の特定’, ‘副作用解消’ によって行う．‘制御構造の抽出’ では実装を入力として細分化モデルの制御構造に対応する制御構造のみを実装した代入文を出力する．

本例では銀行口座システムの顧客登録操作から抽出された図 8.10 の細分化モデルを対象とする．この操作が含む制御構造の遷移条件は SELECT 文の条件節 `!(user).(user : ...)` である．ここではこの遷移条件を P とする．図 8.2 より顧客登録操作の実装は IF 文を一つのみ持ち，その THEN 部の遷移条件は `corFlag=FALSE` である．この遷移条件をモデル変数による表現になおす．corFlag は WHILE の INVARIANT において定義される．図 8.2 の命題 (W1) 中の実装変数をリンク不変条件によりモデル変数に置換すると，users_i.val, userName_i.value, userAddr_i.value がそれぞれ users, userName, userAddr に置換され，図 8.13 の (W1') が得られる．さらに，WHILE ループの終端条件が ii=card(uset) $\leqslant 0$ であることから WHILE ループの終了時に uset は空集合である．よって，IF 文の条件節において命題 (W1') の `users-uset` は `users` となり，図 8.10 の SELECT 文の条件節と等しくなる．

以上により corFlag=FALSE ならば遷移条件 P が真であり，同様に命題 (W2) から corFlag=TRUE ならば遷移条件 P は偽である．このため，細分化モデル (図 8.10) の SELECT 文の THEN 部に対応するのは実装 (図 8.2) の IF 文 THEN 部であると特定される．これにより遷移条件 P において実行されない制御ブロック，すなわち corFlag=TRUE で遷移する制御ブロックが空になる．

次に ‘抽出しない文の特定’ を行う．細分化モデル BankC1 において変化しない細分化

元モデルの大域変数群は users, userName, userAddr, balance, fileOpen である．これらの変数群のリンク変数群 users_i.val, userName_i.value, userAddr_i.value, balance_i.value, users_i.fileOpen, userName_i.fileOpen, userAddr_i.fileOpen, balance_i.fileOpen, が代入禁止実装大域変数 Y_I となる．この代入禁止大域変数で 7.2.4 項にそって抽出しない文の集合 L と未定義変数群 $U(f)$ を求めると，図 8.12(a) の (D2) が抽出しない文の集合 L となり，users_i.val, userName_i.value, userAddr_i.value, balance_i.value が $U(f)$ に加えられる．

次に抽出しない文の集合 L に含まれる文を抽出しないよう拡張した Weiser の手法を制御構造の抽出で得た代入文に適用することで細分化モデル BankC1 の操作に対応した実装の操作を得る．表 8.4 は 2.4.2 項の Weiser の手法に沿ってスライシングを行った例である．表において R は注目する変数 `res_newUser` を基準とした時の直接影響する変数群である．R^1 は WHILE ループにおいてループ変数 ii を基準とした直接影響する変数群である．このスライシングの例では図 8.11 の細分化実装 BankC1_i に示した結果のように (D2) および，`uid:=NID` 以外の代入文が全て抽出される．

表 8.4: 細分化モデル BankC1 についての Weiser の手法適用例

R	S	DEF	REF	PROG
users_i.val, name, addr, uset	○	uset	users_i.val	uset<--users_i.getElements;
users_i.val, name, addr, uset, ii	○	ii	uset	ii<--sizeOfSet(uset);
users_i.val, corFlag, name, addr, uset, ii	○	corFlag		corFlag:= FALSE;
users_i.val, corFlag, name, addr, uset	○		ii	WHILEii>0 DO
users_i.val, corFlag, name, addr, uu, (R^1: uset)	○	uu, uset	uset	uu,uset<--iterSet(uset);
users_i.val, corFlag, uuName, name, addr, uu, (R^1: uset)	○	uuName	userName_i.value, uu	uuName<--userName_i.getVal(uu);
users_i.val, corFlag, uuName, name, uuAddr, addr, (R^1: uset)	○	uuAddr	userAddr_i.value, uu	uuAddr<--userAddr_i.getVal(uu);
users_i.val, corFlag, (R^1: uset)	○		uuName, name, uuAddr, addr	IF uuName=name&uuAddr=addr
users_i.val, corFlag, (R^1: uset)	○	corFlag		THEN corFlag:=TRUE
users_i.val, corFlag, (R^1: uset)				END;
users_i.val, corFlag, (R^1: ii)	○	ii	uset	ii<--sizeOfSet(uset)
users_i.val, corFlag, (R^1: ii)				END;
users_i.val	○		corFlag	IF corFlag=FALSE THEN
users_i.val				VAR ... IN
users_i.val, isEmp	○	isEmp	users_i.val	isEmp<--users_i.isEmpty;
users_i.val	○		isEmp	IF isEmp=TRUE
				THEN
NID	○	NID		NID:=0
NID				ELSE
maxId	○	maxId	users_i.val	maxId<--users_i.getMax;
NID	○	NID	maxId	NID:=maxId + 1
NID				END;
res_newUser	○	res_newUser	NID	res_newUser:=NID;
res_newUser		users_i.val	users_i.val, NID	users_i.add(NID);
res_newUser		userName_i.value	userName_i.value, NID	userName_i.setVal(NID, name);
res_newUser		userAddr_i.value	userAddr_i.value, NID	userAddr_i.setVal(NID, addr);
res_newUser		balance_i.value	balance_i.value, NID	balance_i.setVal(NID, 0);
res_newUser		uid	NID	uid:=NID
res_newUser				END
res_newUser				END

副作用解消

7.2.5 項で示したように副作用解消では操作抽出で得られた代入文を参照部と代入部に分割する．この例では大域変数への代入は res_newUser:=NID である．まず，この代入文を包含する制御構造を参照部に複製する．大域変数への代入を包含する制御構造の遷移条件は corFlag=FALSE である．よって図 8.11 の細分化実装 BankC1_i の (W1) のように制御構造の直前で新たな局所変数 pIF1 に corFlag の値を代入し，代入部に pIF1=FALSE を遷移条件とした制御構造 (Z1) を作る．大域変数への代入文は等価代入文であるため，代入文 res_newUser:=NID において大域変数 res_newUser を (W2) のように局所変数 vIF1 に置換し，(Z2) のように制御構造/*(Z1)*/で res_newUser に vIF1 を代入する．これにより図 8.11 の細分化実装 BankC1_i に示される代入文が得られる．

証明責務最小化

証明責務最小化では証明責務が真であることを保つよう，証明責務を最小化する．この手順は 7.2.6 項で説明したように副作用解消で得られた代入文を細分化実装の代入文 V_I' とし，V_I' に現れる変数と細分化モデルに現れる変数，および，プリミティブな値や型のみで構成された不変条件を仮の不変条件 I_I'' として仮の証明責務のゴールを定め，仮定削減を繰り返すことで行う．上記のように仮の不変条件を定めると I_I'' は users = users_i.val, userName = userName_i.value, userAddr = userAddr_i.value, の論理積となる．この命題 I_I'' について証明責務のゴール $[V_I']\neg[V_A']\neg(I_I'' \wedge u = u')$ を定める．IF 文や WHILE 文を含む実装の証明責務は多くの場合に複雑になり，本例の証明責務では 45 個の命題の論理積になる．ここでは証明責務のゴールの算出に AtelierB を用いた．具体的には命題 I_I'' を不変条件，代入文 V_I' を操作，それらが含む変数定義を細分化元実装の IMPORT 文から抽出することで仮の細分化実装を定め，細分化モデルとの整合性保証の証明責務を生成した．これにより生成される証明責務のゴールは $[V_I']\neg[V_A']\neg(I_I'' \wedge u = u')$ になる．これは $[V_I']\neg[V_A']\neg(I_I'' \wedge u = u')$ が命題 I_I'', 代入文 V_I', および細分化モデルの代入文 V_A' のみで構成されることからも明らかである．

この様に生成されたゴールの例として userName_i.fileOpen = TRUE が挙げられる．これは実装の代入文中で呼び出される操作 uuName <-- userName_i.getVal(uu) の事前条件から生じる命題である．この命題を真にするのに必要な命題は実装の不変条件 ((fileOpen = TRUE) =>(userName_i.fileOpen = TRUE)) とモデルの事前条件 fileOpen = TRUE の 2 つであり，これが仮定削減結果となる．同様の仮定削減を他のゴールにも適用することで図 8.11 の BankC1_i のように不変条件が定まる．また，WHILE ループでは card(uset-xx)+1 <= ii のようなゴールが生成され，仮定削減により WHILE ループの不変条件から ii = card(uset) などの命題が抽出される．今回の例では WHILE ループ内の代入文が細分化元のループ内の代入文と等しいため，ループの不変条件も細分化元と等

しくなる．以上により図 8.11 で示した細分化実装 BankC1_i と実装依存モデル BankC1_d が生成される．

8.3.3　部品登録

7.3 節で述べたように部品登録では‘階層構造の構築’と‘モデル実装間変数対応付け’を行う．

‘階層構造の構築’では 7.3.1 項で述べたように事前条件以外が一致する部品間に階層構造を構築する．この例で扱う銀行口座システムとグループ管理システムから生成される細分化モデルはいずれも不変条件と代入文が異なる．そのため，空の部品リポジトリに銀行口座システムとグループ管理システムの部品群を登録しても階層構造は生じない．階層構造の構築は 7.3.1 項で例示したため，ここでは省略する．

‘モデル実装間変数対応付け’では 7.3.3 項で述べたように細分化実装のリンク不変条件から実装依存モデルと細分化実装間の変数間対応を生成する．図 8.11 の細分化実装 BankC1_i の不変条件から‘モデル変数 = (実装変数の式)’という命題を 3 つ持つ．ここで，これらの等式からモデル変数 users, userName, userAddr に対して実装変数 users_i.val, userName_i.value, userAddr_i.value がそれぞれ等価である．これによりモデル実装間変数対応 (users, users_i.val), (userName, userName_i.value), (userAddr, userAddr_i.value) が得られる．この変数間対応を部品と共に部品リポジトリに格納する．

8.4　MSSS の適用例

8.4.1　概要

本節では MSSS の適用例として，本節ではマイレージ付きレンタカーシステムの車両返却を生成する．生成に用いる部品リポジトリには銀行口座システムとグループ管理システムから生成された部品群が登録されているとする．図 8.5 に示したようにレンタカーシステムの車両返却は‘銀行口座システムの預金預入操作’から生成される部品 C_{B5} と‘グループ管理システムのグループ脱退操作’から生成される部品 C_{G5} を再利用して合成される．部品 C_{B5}, C_{G5} を図 8.14 に示す．図 8.15 にレンタカーシステムの車両返却から生成される細分化モデルを示す．図 8.14 の細分化モデル BankC5 と図 8.15 の細分化モデル RentC7 において変数名を maxBalance から maxPoint に，balance を points に，amount を point に置換することで字面が一致する．この事から細分化モデル BankC5 と細分化モデル RentC7 の機能が等しい事が分かる．同様に細分化モデル GroupC5 と細分化モデル RentC6 の字面が一致し，その機能が等しいことが分かる．

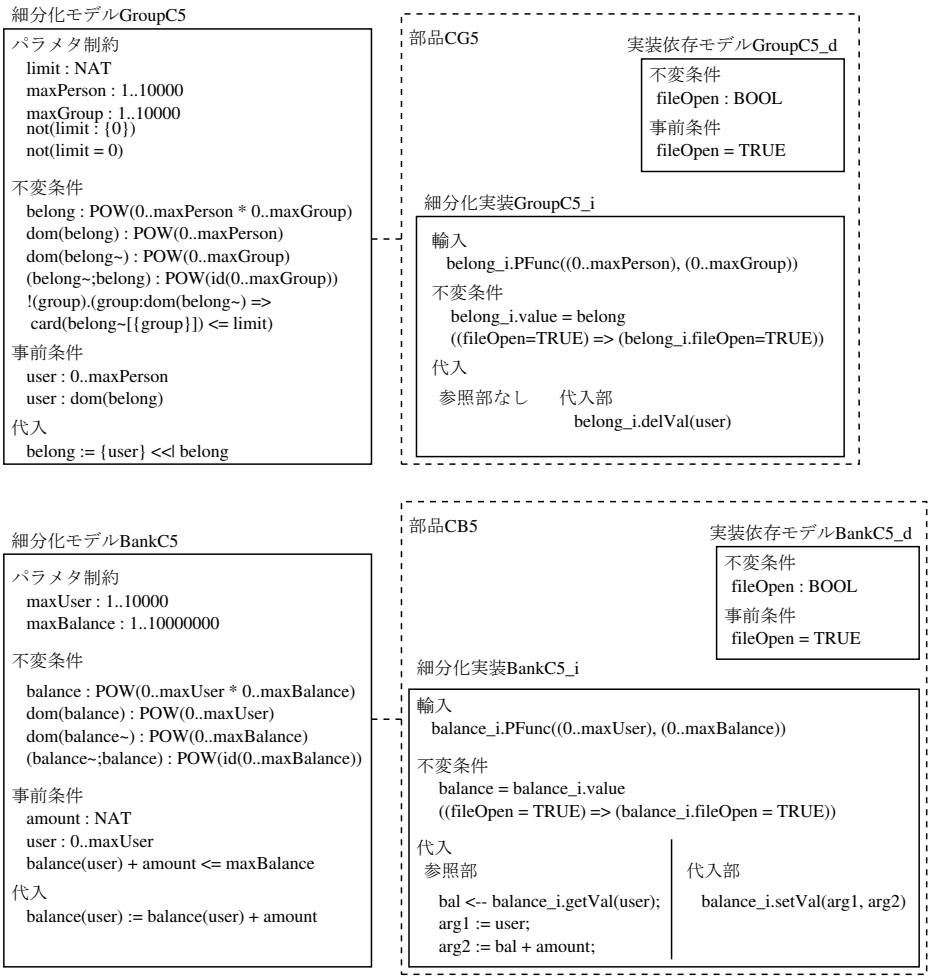


図 8.14: レンタカーシステムの車両返却で再利用する部品

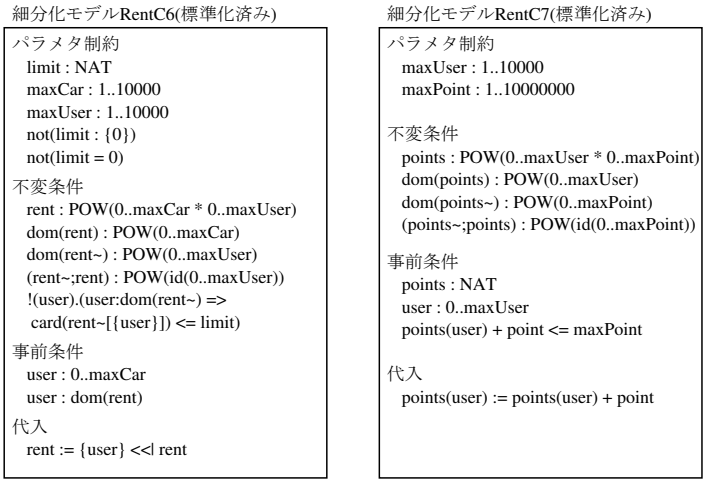


図 8.15: 車両返却から生成される細分化モデル

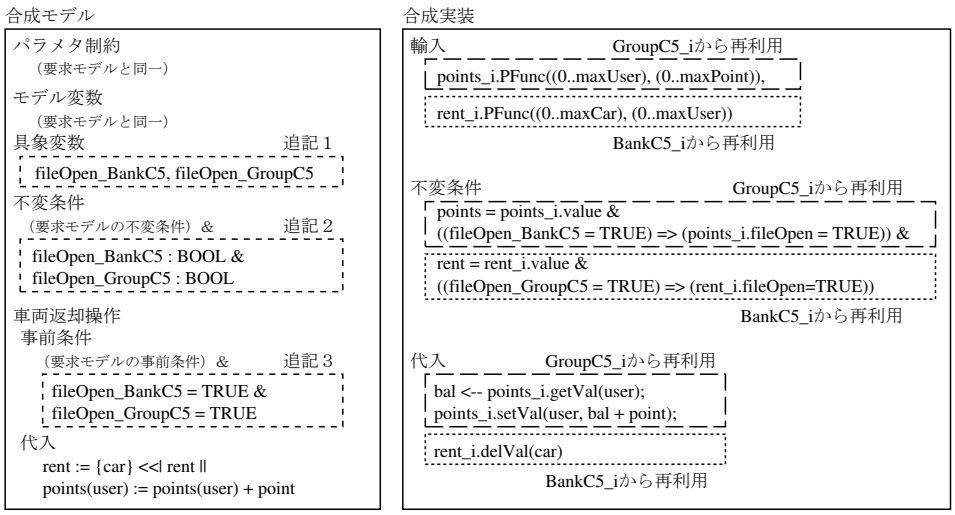


図 8.16: 車両返却についての合成モデルと合成実装

部品 C_{B5}, C_{G5} の実装依存モデルと細分化実装を MSSS の手順に沿って結合することで図 8.16 の合成モデルと合成実装が得られる．これによって得られた合成実装の操作 returnCar からは 14 個の証明責務が生成され，全て真であることを証明できた．ただし，合成モデルは入力モデルに存在しない実装依存の変数 `fileOpen_BankC5` と `fileOpen_GroupC5` を持つため，これらの変数に対する初期化処理を追加する必要がある．

以降の節では上述の操作 returnCar の合成手順を 6 章で説明したモデル充足ソフトウェア合成手順に基づき例示する．

8.4.2　モデル細分化

ここでは本例題で合成する操作であるレンタカーシステムの車両返却のモデル細分化について説明する．ただし，モデル細分化の詳細な適用例は 8.3.1 項ですでに挙げたため，ここではモデル細分化にあたり注意が必要な点と生成される細分化モデルの機能について説明する．車両返却は図 8.4 の returnCar で示したように操作の引数として車両 (car), 顧客 (user), マイレージ (point) を受け取り，貸出 (rent) から与えられた車両を取り除き，顧客のマイレージに与えられたマイレージを加算する．この際，加算されたマイレージはマイレージの上限 (maxPoint) を超えないことを事前条件で定めている．returnCar の代入は rent と points に対する同時代入である．ここで rent と points は等価な変数では無いため，図 8.15 のように rent を注目する変数とした細分化モデル RentC6 と points を注目する変数とした細分化モデル RentC7 が生成される．制約条件抽出で注意が必要な点として limit の扱いが挙げられる．不変条件には `!(user)` から始まる rent と limit からなる述語論理が存在する．この limit はモデルパラメタであるため 5.5 節に述べたように操作に limit が登場しない RentC6 において limit を含む制約条件が抽出される．以上により図

8.15 の細分化モデルが生成される．

8.4.3　マッチング

ここでは部品リポジトリに図 8.14 の細分化モデル BankC5, GroupC5 が格納されているとして図 8.15 の RentC6, RentC7 にマッチする部品を文字列一致により判定する．6.2.2 項に示したようにマッチングは細分化モデル間で変数を置換しながら文字列を比較することで行う．

図 8.14 の GroupC5 と図 8.15 の RentC6 において GroupC5 の変数名を登場順に RentC6 の変数名に書き換えると maxPerson が maxCar に，maxGroup が maxUser に，belong が rent に，user が car に置換される．このときパラメタ制約，不変条件，事前条件，代入文が完全一致する．また，図 8.14 の BankC5 と図 8.15 の RentC7 において BankC5 の変数名を登場順に RentC7 の変数名に書き換えると maxBalance が maxPoint に，balance が points に，amount が point に置換される．このときパラメタ制約，不変条件，事前条件，代入文が完全一致する．

以上により細分化モデル RentC6 に対して部品 C_{G5} が，細分化モデル RentC7 に対して部品 C_{B5} が再利用される．

レンタカーシステムの細分化モデルのうち，車両貸出から生成される細分化モデル RentC5 だけはマッチする部品が存在しない．図 8.17 は細分化モデル RentC5 とグループ管理システムから生成された部品 C_{G3} の細分化モデル GroupC3 である．GroupC3 の変数名を登場順に RentC5 の変数名に置き換えると maxPerson は maxCar に，maxGroup は maxUser に，belong は rent に，person は car に，group は user に置換される．このときパラメタ制約，不変条件，代入文が完全一致する．しかし，RentC5 に対して GroupC3 がマッチするためには RentC5 の事前条件が GroupC3 の事前条件を包含しなければならないが，この例では逆に GroupC3 の事前条件が RentC5 の事前条件を包含しておりマッチしない．RentC5 に対して GroupC3 がマッチしないことは RentC5 の事前条件の命題数 n_K が 5 であるのに対して GroupC3 の事前条件の命題数 n_A が 6 であり，$n_K < n_A$ であることから判定される．RentC5 にマッチする部品が存在しないため合成されるソフトウェアは不完全なものとなる．ただし，8.4.6 項の ‘不完全なソフトウェアへの対応’ で述べるように最終的には細分化モデル RentC5 に対して部品 C_{G3} が再利用される．

8.4.4　変数名統一

8.4.3 項のマッチングでは図 8.15 で示したレンタカーシステムの細分化モデル RentC6 に対してそれぞれ図 8.14 の部品 C_{G5} がマッチした．この節では 6.2.3 項で示した変数名統一手順に沿って部品 C_{G5} の実装依存モデルと細分化実装の変数名を RentC6 の変数名

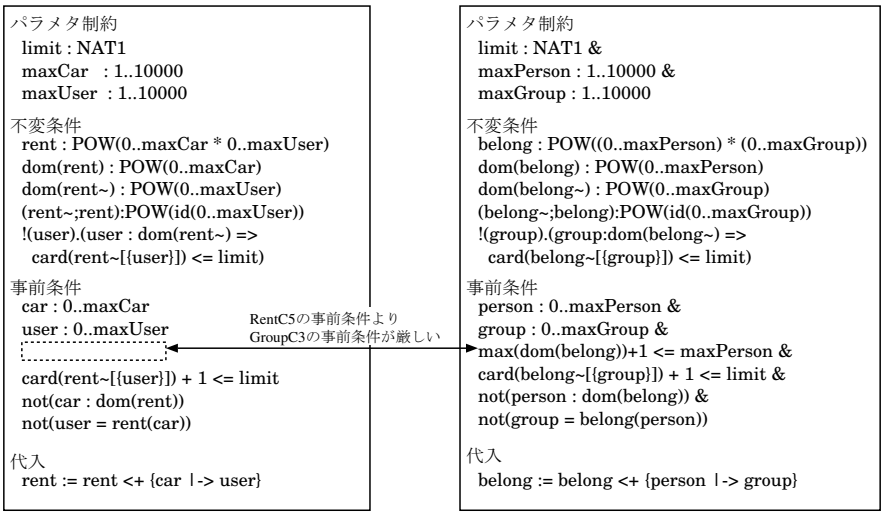


図 8.17: 細分化モデル RentC5 と細分化モデル GroupC3 間のマッチング

に置換する．図 8.18 に部品 C_{G5} と部品 C_{G5} に変数名置換を適用した結果を示す．

モデル実装間変数対応付けとして部品 C_{G5} は (belong, belong_i.value) を持つ．まず，部品 C_G5 の細分化モデル GroupC5 の変数名を置換する．これは細分化モデル GroupC5 の変数名を登場順に細分化モデル RentC6 に書き換えることで行う．これにより，limit, maxPerson, maxGroup, belong, user がそれぞれ limit, maxCar, maxUser, rent, car に置換される．次に実装依存モデル GroupC5_d の変数名を置換する．実装依存モデルの変数名置換は細分化モデルの変数名置換と同様に置換を行った上で，細分化モデルに存在しない変数名に対して部品名をプレフィックスとすることで部品毎に固有の変数名を与える．GroupC5_d では変数 fileOpen が GroupC5 に存在しない変数であるため，fileOpen を GroupC5_fileOpen に置換する．また，細分化実装に現れるモデル変数に対しても実装依存モデルの変数名置換を適用する．次に細分化実装の変数名を置換する．細分化実装の変数名は‘輸入で生じる変数’，‘細分化実装の CONCRETE_VARIABLES で宣言される変数’，‘局所変数’ に分類される．‘輸入で生じる変数’ には belong_i.val が該当する．部品 C_{G5} はモデル実装間変数対応付けとして (belong, belong_i.value) を持ち，モデル変数 belong は rent に置換されるため，モジュールの別名 belong_i を rent_i に置換する．これにより，belong_i.val は rent_i.val になる．部品 C_{G5} は局所変数を持たないため，局所変数名の置換は生じない．以上により部品 C_{G5} の実装依存モデルと細分化実装は図 8.18 の部品 C'_{G5} のようになる．

同様に細分化モデル RentC7 にマッチする図 8.14 の部品 C_{B5} に変数名置換を適用すると図 8.18 の部品 C'_{B5} のようになる．

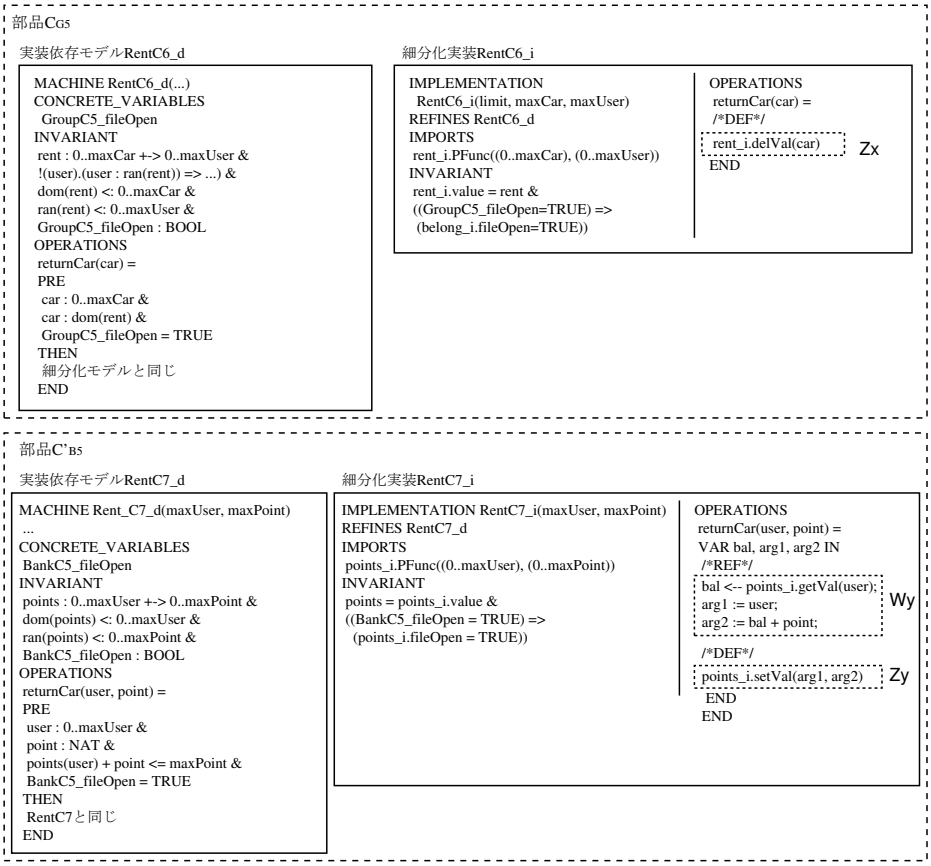


図 8.18: 部品 C_{G5}, C_{B5} の変数名置換結果

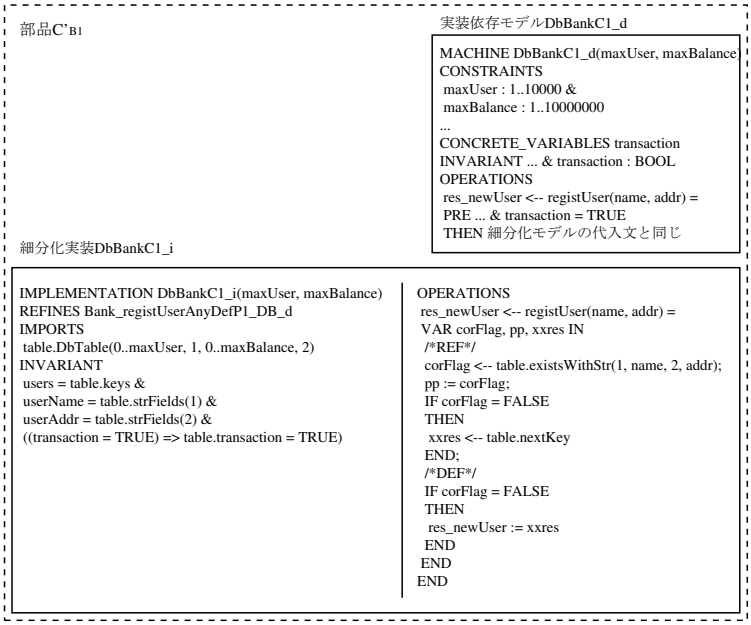


図 8.19: 細分化モデル BankC1 のデータベースによる実装

8.4.5　部品選択

図 8.5 で示したように要求モデルの各検索キーに対して銀行口座システムとグループ管理システムから生成した部品群はたかだか一つずつマッチする．このため，この例では実装方法は一通りしか存在せず，利用者による実装方法の選択は生じない．本節では実装方法の選択を例示するために 8.1.1 項で紹介した銀行口座システムとは実装方法の異なる銀行口座システムが部品化されてリポジトリに格納されていると仮定する．8.1.1 項で紹介した銀行口座システムで実装に用いたモジュール群 Set, StrPFun, PFun は情報をファイルに格納するが，ここで仮定する銀行口座システムは情報をデータベースに格納するモジュール DbTable を用いて実装される．モジュール DbTable を用いた部品の例を図 8.19 に示す．図 8.19 の部品 C'_{B1} は図 8.11 の部品 C_{B1} と同じく細分化モデル BankC1 を仕様として持ち，モジュール DbTable を用いて実装している．

まず，部品検索で得られた部品間で 6.3.2 項の選択可否判定を行う．表 8.5 はモデル変数の部品 C_{B1}, C'_{B1}, C_{B2}, C_{G3} の実装における型を表す．例えば，先のマッチングにおいて部品 C_{G5} は細分化モデル GroupC5 の変数名 belong を rent に置換することで検索キー RentC6 にマッチした．よって，部品 C_{G5} における rent の型は belong のリンク変数 belong_i.val の型，モジュール PFun となる．表 8.5 から部品 C_{B1}, C_{B2}, C_{G3}, C_{G5} 間で型の不一致は無いためこれらの選択可否判定は真である．また，データベースを用いた部品 C'_{B1} については C'_{B1}, C_{G3}, C_{G5} 間で型の不一致は無いため C'_{B1}, C_{G3}, C_{G5} の選択可否判定は真である．しかし，部品 C'_{B1}, C_{B2} 間では users, userName, userAddr の型が一致しないため部品 C'_{B1} と C_{B2} の選択可否判定は偽になる．

表 8.5: 部品 C_{B1}, C'_{B1}, C_{B2}, C_{G3} 間におけるモデル変数の型

部品	users	points	userName	userAddr	rent
C_{B1}	Set.val	-	StrPFun.value	StrPFun.value	-
C'_{B1}	DbTable.keys	-	DbTable.strFields(1)	DbTable.strFields(2)	-
C_{B2}	Set.val	PFun.value	StrPFun.value	StrPFun.value	-
C_{G3}	-	-	-	-	PFun.value
C_{G5}	-	-	-	-	PFun.value

これにより，表 8.5 から決定される選択可能な部品の組み合わせは $\{C_{B1}, C_{B2}, C_{G3}, C_{G5}\}$ と $\{C'_{B1}, C_{G3}, C_{G5}\}$ になる．

次に利用者による実装方法の選択を例示する．6.3.3 項で示したように実装方法の選択は要求モデルの各変数について選択可能なモジュールを提示し，利用者がそれらを選択することによって行う．表 8.5 の例ではモデル変数 users に対して Set と DbTable が，モデル変数 userName, userAddr に対して StrPFun と DbTable が提示される．ここで，利用者が users に対して Set を選択すると部品 C_{B1} が選択されるため，先の選択可能な部品の組合わせのうち，$\{C_{B1}, C_{B2}, C_{G3}, C_{G5}\}$ が選択される．逆に users に対して DbTable を選択すると部品の組み合わせ $\{C'_{B1}, C_{G3}, C_{G5}\}$ が選択される．

8.4.6　部品結合

本節では要求モデルの細分化モデル群に対して部品の組 $\{ C_{B1}, C_{B2}, C_{B3}, C_{B4}, C_{B5}, C_{G5} \}$ が部品選択で選択されたと仮定して部品結合を例示する．部品結合は 6.4 節で示したように出力モデル，出力実装の制約条件の合成，操作の合成によって出力モデルと出力実装を生成する．さらに，出力モデルと出力実装から B Method の枠組みで証明責務を生成し，証明責務が偽になる場合には不完全なソフトウェアへの対応を行う．以下の節では部品結合の各手順を例示する．

制約条件の合成

6.4.1 項で述べたように要求モデルの不変条件を I_K 変数名の置換を適用した実装依存モデルの不変条件を I'_{Di} としたとき，出力モデルの不変条件 I_A は $I_A = I_K \bigwedge I'_{Di}$ と表される．図 8.18 の実装依存モデル RentC6_d と RentC7_d にこれを適用すると図 8.20 の合成モデルの不変条件が得られる．この図において I_x が RentC6 の不変条件であり，I_y が RentC7 の不変条件である．パラメタ制約とプロパティ制約についても同様に得られる．

変数名の置換を適用した細分化実装の不変条件群を I'_{Ii} としたとき，出力実装の不変条件 I_I は $I_I = \bigwedge I'_{Ii}$ と表される．図 8.20 の合成実装の不変条件は図 8.18 の細分化実装 RentC6_i と RentC7_i の不変条件を合成したものである．また，各細分化実装の輸入 (IMPORTS) 節を重複を省いて書き連ねることで合成実装の輸入を生成する．

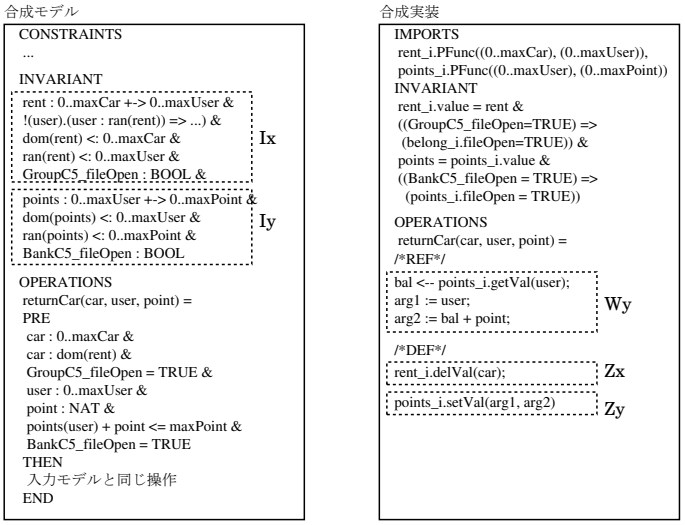


図 8.20: 車両返却 (returnCar) の合成結果

操作の合成

6.4.2 項で示したように部品群の実装依存モデルの事前条件群 Q'_{Di} としたとき, 出力モデルの事前条件は $\bigwedge_{i=1}^{l} Q'_{Di}$ から操作中で宣言される変数を取り除いたものである．図 8.4 に示した要求モデルにおいて操作 returnCar は ANY の様な変数宣言を持たないため，図 8.18 の細分化実装 RentC6_d と RentC7_d の事前条件を単純に論理積で結合すれば良く，図 8.20 のように事前条件が得られる．

また，部品の細分化実装の代入文のうち非決定的値生成操作について検索されたものを (W_p^d, Z_p^d)，非決定的値参照操作について検索されたものを (W_q^r, Z_q^r) としたとき，合成実装の代入文は式 (8.1) のようになる．ここで， $W_q^{\prime r}$, $Z_q^{\prime r}$ は Z_p^d において戻値 res_x に代入される式 E としたときに， W_q^r, Z_q^r の引数 x を式 E に置換したものである．

$$V_I = W_1^d;\ \ldots;\ W_{m_d}^d;\ W_1^{\prime r};\ \ldots;\ W_{m_r}^{\prime r};\ Z_1^{\prime r};\ \ldots;\ Z_{m_r}^{\prime r} \tag{8.1}$$

図 8.18 の細分化実装 RentC6_i と RentC7_i の代入文はどちらも非決定的値参照操作であるため非決定的値生成操作の戻値にあたる引数の置換は不要である．よって，図 8.20 のように単純に W_y;Z_x;Z_y と結合することで合成実装における操作 returnCar の代入文が得られる．

また，図 8.21 の細分化実装の代入文を合成することで図 8.22 のようにレンタカーシステムの顧客登録操作の実装が得られる． 図 8.21 はレンタカーシステムの顧客登録操作 (addUser) についての部品検索で得られた部品群である．これらの部品群のうち，部品 C'_{B1} は非決定的値生成操作から得られた検索キーにマッチし，C'_{B1}, C'_{B1}, C'_{B1} は非決定的値参照操作から得られた検索キーにマッチする．

部品 C'_{B1} の代入部 Z_1 において戻値 `res_newUser` に対して `BankC1_vIF1` を代入するた

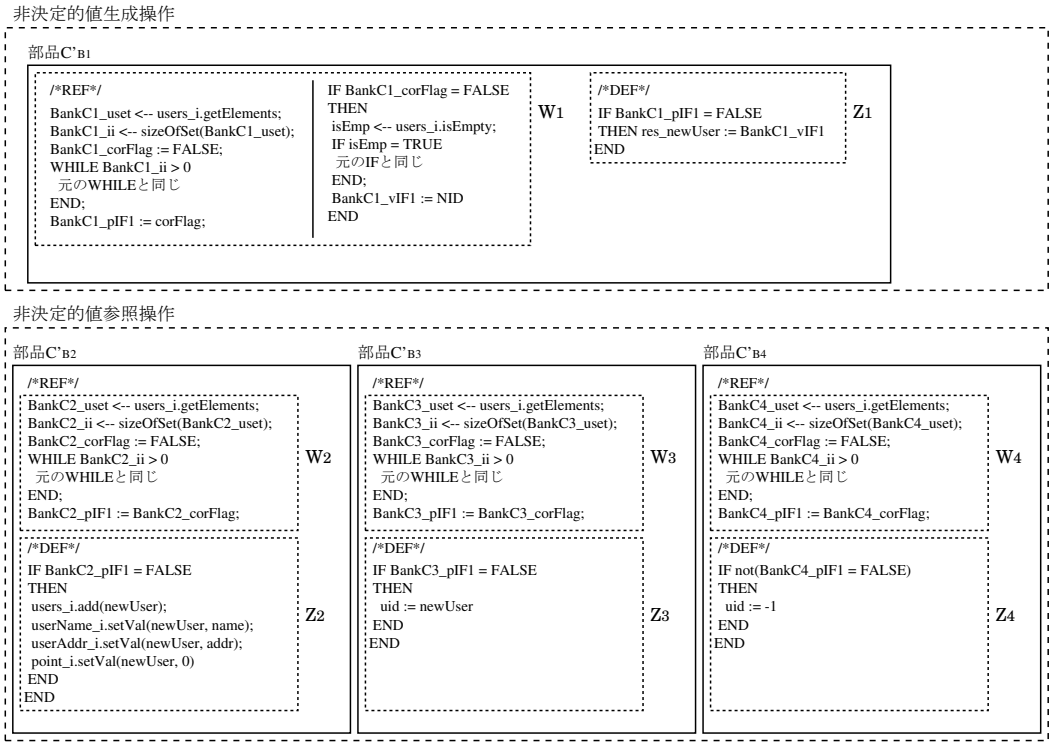


図 8.21: 顧客登録操作に対してマッチした部品群の変数名置換結果

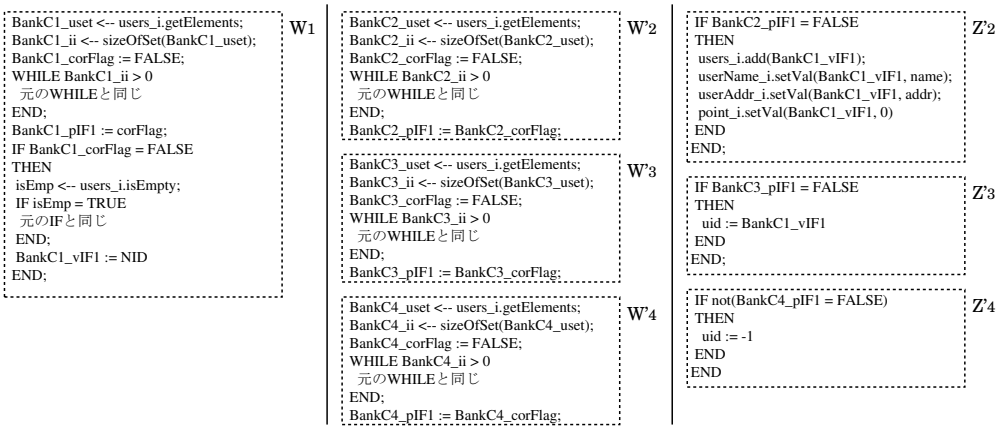


図 8.22: 顧客登録操作 (addUser) の合成結果

め，部品 C'_{B1}, C'_{B1}, C'_{B1} の参照部 W_2, W_3, W_4 と代入部 Z_2, Z_3, Z_4 において newUser を BankC1_vIF1 に置換した代入文をそれぞれ参照部 W'_2, W'_3, W'_4，代入部 Z'_2, Z'_3, Z'_4 とする．これらを W_1; W'_2; W'_3; W'_4; Z'_2; Z'_3; Z'_4 と結合することで図 8.22 の代入文が得られる．

不完全なソフトウェアへの対応

6.5 節で述べたように MSSS で得られる出力ソフトウェアは証明責務が偽である可能性があり，証明責務が偽である場合には利用者による部品の追加などの対処が必要になる．

本章では銀行口座システムとグループ管理システムから得られた部品群を再利用してマイレージ付きレンタカーシステムを合成する例を示したが，これによって得られる合成モデルと合成実装は以下の点で証明責務が偽になる．

1. 要求モデルに存在しないモデル変数 fileOpen が合成モデルに追加された．
2. 車両貸出 (rentCar) に対して部品が割り当てられなかった．

図 8.20 のように合成モデルには要求モデルに存在しない変数 GroupC5_fileOpen, BankC5_fileOpen が存在する．証明責務が真になるためには変数を初期化しなければならず，利用者がこれらの変数の初期化を合成モデルに追記して MSSS を再度適用する．この際，初期化以外の操作には変化が無いため，MSSS は初期化についてのみ行えば良い．本例の場合，GroupC5_fileOpen := FALSE, BankC5_fileOpen := FALSE という初期化を追加し，MSSS を再度適用すると，実装の初期化として GroupC5_fileOpen := FALSE と BankC5_fileOpen := FALSE が追加される．

また，本例では 8.4.3 のマッチングで述べたように要求モデルから得られた細分化モデル RentC5 に対して部品が割り当てられなかった．そのため，利用者は RentC5 に対して部品を与える必要がある．6.5 節で述べたように利用者が部品を与える際には ‘部品の仕様の提示’，‘類似部品の提示’，‘記述の自動生成’ の補助が行われるが，この例では ‘類似部品の提示’ により問題が解決する．類似部品の提示はマッチングにおいて事前条件のみがマッチしない部品を提示し，その部品の事前条件を検索キーの事前条件に書き換えた上で部品の証明責務を再度証明する．RentC5 の例では図 8.17 の様に部品 C_{G3} の細分化モデル GroupC3 が検索キーに無い事前条件 max(dom(belong)) <= maxPerson - 1 を持つ為に生じる．そのため，部品 C_{G3} の細分化モデルと実装依存モデルからこの事前条件を取り除いた部品 C'_{G3} を生成し，B Method の枠組みで利用者が証明責務の証明を行う．この時，証明責務は真になり，RentC5 で利用可能であることが明らかになるため，これを部品リポジトリに追加した上で操作 rentCar について MSSS を再度適用する．

本例では上記のように不完全なソフトウェアへの対応を行うことで証明責務が真である合成モデルと合成実装が得られる．

第9章

考察

本章では8章の手法適用例を評価すると共に手法の適用性，信頼性，計算量，作業量を定性的に評価する．

9.1 適用性

本節ではMSSSフレームワークの適用性を‘適用可能な問題領域’, ‘MSFCの再利用性’に分類して考察する．なお，ここではコストについては論じず，合成手法の適用可能な問題領域や部品の汎用性と機能性のみに着目して考察する．手法適用に要するコストについては9.4節で考察する．

9.1.1 適用可能な問題領域

2.3.1項に示したように経験則に基づく自動コード生成は数学的な裏付けが無くとも人間が自由にパターンを与える事が出来るため，現実の問題に対して高い適用性を持つ．これに対して，3.3.1項で述べたようにMSSSフレームワークの適用可能な問題領域はB Methodが適用可能な問題領域に限定される．このため，ユーザインタフェースや実時間制御や非同期処理などのB Methodが本質的に適用できない問題群はMSSSで自動生成することは出来ない．しかし，B Methodが対象とする問題領域が実運用システムの開発においても有意であることはB Methodが パリメトロ14号線などの開発実績を持つ事からもうかがえる．

上述のようにMSSSフレームワークの適用可能な問題領域は原理的には既存の自動コード生成手法よりも狭い．しかし，現実的には既存の自動コード生成手法が利用されている問題領域は限定的である．これは経験則に基づく自動コード生成を適用するためには問題領域毎に部品リポジトリを整備する必要があり，この整備に多くの時間と労力を必要とするためである．これに対してMSSSフレームワークは図3.2で示したように既存ソフトウェアから MSFCを生成する事で迅速に部品リポジトリを構築できる．このため，

MSSS フレームワークは原理的には適用可能な問題領域が狭いが部品リポジトリの整備性を考慮すると既存の自動コード生成手法より多くの問題領域に適用できると考える．

9.1.2　MSFC の再利用性

2.4 節で述べたように本稿では部品の汎用性と機能性をまとめて再利用性と呼ぶ．2.4.1 項で述べたように部品の粒度に注目すると汎用性と機能性はトレードオフの関係にありるが，MSSS フレームワークのように機能を機械的に合成するならば 1 部品の機能性は問題では無い．これにより，MSSS は部品の粒度を細粒度にすることで再利用性を向上している．また，2.4.3 項で述べたように機能を機械的に合成するためには数学的仕様の付加が有効であるが膨大な部品群に数学的仕様を与えることは部品リポジトリの生成を困難にする．MSSS フレームワークではこの問題を MSFC 生成により解決している．

本手法では 4.2.1 項で述べたように以下のように部品の粒度を定めた．

1. 1 部品では 1 変数のみが変化する．
2. ANY 文は非決定的値の戻値への代入のみを持つ．
3. SELECT 文の条件群は互いに非決定的である．

この様に部品の粒度を細粒度化することで部品の汎用性が向上し，‘ANY 文で定義される非決定的値は同じであるが，計算部分が異なるソフトウェア間の部品共有’ などが可能となる．MSSS フレームワークでは部品を既存ソフトウェアから生成するため，部品の粒度を決定するにあたっては ‘モデルと同様の細分化を実装に対しても行えること’ が重要である．このため，MSSS フレームワークでは 3.3.2 項のようにモデルと実装の文法を制限している．特に ANY 文を非決定的値生成操作と非決定的値参照操作に分割するためには ‘大域変数への代入で参照される式は局所変数に格納可能である’ 事が必要である．ANY 文を含む操作の実装において大域変数への代入で参照される式から非決定的値を特定して戻値に格納することで非決定的値生成操作の実装が得られる．また，細粒度の条件として ‘SELECT 文の条件群が非決定的である事’ が定められているのは SELECT 文をそれ以上細かく分割できる保証が無いためである．これは，決定的な条件群は条件に対して動作が一意に定まるため実装においても条件毎に分割できるが，非決定的な条件群では重なりを持つ条件において複数の実装が考えられるため実装の条件を一意に分割できないためである．

8 章の手法適用例では銀行口座システム，グループ管理システムという本来はドメインの異なるソフトウェアから MSFC を生成し，その部品群を再利用してマイレージ付きレンタカーシステムを自動生成した．この様に異なるドメイン間で横断的に部品を再利用できるのは MSFC の仕様が数学的であり，機能名や部品名に依存しない部品検索が可能なためである．

SELECT 文などの制御構造に包含される代入文を細分化する際，制御構造のみの部品と代入文のみの部品に分割できればより部品の粒度を細かくすることができる．しかし，本手法では部品の制御構造が空になることを許していない．このため，モデルの代入文が制御構造に包含されるならばその細分化モデルの代入文も制御構造に包含される．この様に制御構造と代入文を分割しないことには以下の 2 つの理由が挙げられる．

1. 制御構造だけを持つ部品は B Method の枠組みで信頼性を保証できない．
2. 制御構造から抽出した部品の事前条件が厳しく再利用性に乏しい．

2.5.3 項で示したように B Method の証明責務は $[V]I$ のような最弱事前条件で表されるが，V が制御構造だけを持つ，すなわちシステムの状態を変化させない場合には証明責務は常に真になる．そのため，制御構造だけを持つ部品は B Method の枠組みで信頼性を保証できない．また，遷移条件 p で代入文 f が実行されるとき，その証明責務は $p \Rightarrow [f]I$ の様な形を持ち，$p \Rightarrow [f]I$ が真であるとき $[f]I$ が真であるとは限らない．そのため，代入文 f を制御構造から抽出する際には遷移条件 p を操作の事前条件とする必要があるが，この様な部品は事前条件が厳しすぎて再利用性が乏しい．

以上の様に本手法では部品の信頼性が保証できる範囲内で部品を細粒度化することで部品の再利用性を向上させている．

9.1.3　現在のソフトウェア開発における位置づけ

本研究は形式手法におけるソフトウェア開発を自動化することで，形式手法適用コストを低減することを目的としている．この目的は従来から形式手法を用いてきた高信頼ソフトウェア開発のコスト低減だけでなく，それまで形式手法を適用することが困難であったソフトウェア開発へ形式手法を適用することも本研究の目標としている．

一方で，3.3.1 項に挙げたように，MSSS フレームワークで開発可能なソフトウェアは以下のように制限される．

1. 全ての状態は集合論と述語論理で記述できる．
2. 操作は戻値を返すか，システムの状態を変化させる．
3. 操作開始から完了までの処理時間を要求や制約として与えない．
4. 各操作は同期処理であり，同時には実行されない．また，開始した操作は割り込みを受けずに終了する．

これらの制約から MSSS フレームワークは GUI や I/O の実装には不向きである．GUI については集合論と述語論理だけでは見た目を仕様に記述することが困難であり，現在のソフトウェア開発で求められる洗練されたユーザ体験の実現を自動化できないためである．また，I/O を集合論と述語論理で記述する場合，‘I/O で入力されるデータ，出力

されるデータに特定の情報が含まれること’ を記述することは容易であるが，その書式までを指定することは困難である．

我々はこの問題はソフトウェア開発の分業化で解消されると考えている．ソフトウェア開発の分業化については 2.2 節で紹介したが，代表的な例である MVC アーキテクチャではデータ表現，ユーザインタフェース，制御を分離して開発する．MVC アーキテクチャの中で MSSS 手法を用いるならば，制御を MSSS により実装し，データ表現とユーザインタフェースはそれぞれに適した言語で実装することが考えられる．具体的にはデータ表現とその入出力を行う操作を命令型言語で記述し，その仕様として B Method のモデルを与えることで I/O とデータ表現を B Method のモジュールとして定義する．これらを用いてシステムの状態を制御する操作を B Method のモデルで記述し，MSSS 手法により操作を実装する．さらに，これらを呼び出す制御プロセスにおいて，ユーザインタフェースとのつなぎ込みを行うことで，１つのシステムとして動作させる．なお，I/O についての制限は MSSS 手法が基盤としている B Method に起因する制限であり，B Method の事例の一つである鉄道システムの開発においても，入出力は Ada で記述されている [32]. この様に，MSSS は既存のソフトウェア開発を置き換えるものではなく，既存のソフトウェア開発の中で制御の実装に利用される 1 手法と見ることができる．

また，近年ではテスト駆動開発 (TDD) の流れから，テストデータの恣意性の排除と，より，仕様を意識したテスト記述が可能な振る舞い駆動開発 (BDD) が注目されている．BDD では集合論と述語論理で関数仕様を記述するため，これが普及することで数学を基盤とした仕様記述が，より一般的な開発でも可能になると期待できる．さらに，TDD, BDD においてもテスト仕様の記述は実装前に行うことが求められている．この点から，BDD におけるテスト記述者が MSSS のモデルを記述することで，既存開発への MSSS の適用が容易になることが期待できる．

9.2　信頼性

本節では MSSS の信頼性と MSSS で利用される部品を生成する MSFC 生成の信頼性についてそれぞれ考察する．

9.2.1　MSSS の信頼性

MSSS は健全性の高い部品検索で MSFC を選択し，それらを結合することでソフトウェアを合成する．ここで，リポジトリ内の MSFC は 4.3 節のように証明責務の証明により信頼性が保証されていると仮定されるため，そのため，MSSS で得られるソフトウェアの信頼性は部品選択，部品結合の信頼性で定まる．以下の節ではそれぞれの信頼性について考察する．

部品選択の信頼性

MSSS における部品選択は 6.2 節の部品検索と 6.3 節の部品選択により行われる．部品検索と部品選択の信頼性は以下のようにまとめられる．

1. 部品検索は従来のコード生成手法における検索やパターン判定に比べて高い健全性を持つ．
2. 部品選択は証明責務が偽になることが明らかな部品の組合わせを排除する．ただし，常に証明責務が真になるとは限らない．
3. 数学的に判定できない部品選択の妥当性は部品選択において利用者が実装方法を選択することで保証する．

従来のコード生成手法は意味付けを部品名や機能名に頼っており，名前付けが非妥当であったり，名前の解釈を誤った場合には正しいコードが生成されない．これに対して 6.2 節の部品検索では数学的な仕様であるモデル間で数学的に判定を行うため，非妥当な名前付けや名前の誤解釈による誤りが生じない．ただし，仕様間の数学的判定は健全性が高い反面で完全性の保証には仕様間の判定に定理証明を必要とし計算量爆発を招く．MSSS ではこの問題を細分化モデルの字面統一により仕様間の判定を字面一致で可能にすることで解決している．この部品検索の計算量低減は 9.3 節で説明する．

6.3 節の部品選択では選択可否判定により合成ソフトウェアの証明責務が偽になる事が明らかな組み合わせを排除することで合成ソフトウェアの信頼性を向上している．6.3.2 項で述べたように選択可否判定では型の文字列一致判定のみを行っており，型以外の制約条件が矛盾する可能性がある．選択可否判定に定理証明を用いることで合成ソフトウェアの証明責務が真になることを保証できるが，先行する定理証明に基づいた自動コード生成手法が計算量の問題で実用化されていないことを考えれば計算量の低減を重視することは妥当と言える．

部品検索と選択可否判定はどちらもソフトウェアの数学的な側面だけで判定を行うため，部品の組み合わせの信頼性は証明責務で保証できるものに限定される．この場合，証明責務は真だが非妥当な組み合わせが生じうる．例えば，8.4.5 項の部品選択可否判定例では変数 users などをデータベースに保存する部品 C'_{B1} と変数 rent をファイルに保存する部品 C_{G3}, C_{G5} の選択可否判定が真であることを示した．しかし，多くの場合には users と rent の両方がデータベースに格納されることが妥当である．MSSS では 6.3.3 項の ‘利用者による実装方法の選択’ で user, rent の実装に用いるモジュールを選択することにより妥当な部品の組み合わせを用意に選択することができる．

部品結合の信頼性

互いに矛盾しない部品を再利用して得られるソフトウェアが高信頼であるためには, 再利用手順自体が高信頼である必要がある. 6.4 節の部品結合は制約条件の結合と操作の結合に分けられる. 本手法では出力モデルと出力実装のプロパティ制約や不変条件を部品の制約条件の論理積で生成する. これにより部品群が互いに矛盾する制約条件を持たないならば, それらを論理積で結合した制約条件も矛盾しない. 6.4.2 項の操作の結合では要求モデルから得た細分化モデルを元通りに組み直すのと同様にその細分化モデルを仕様として持つ細分化実装をモデル細分化手順を逆に辿るように結合する. 部品検索で得られた部品群が要求モデルから得られた細分化モデルを仕様として持つことは 6.2 節の部品検索において式 (6.5) が成り立つことから保証される. また, 細分化モデルと同様に細分化実装の操作を合成するためには 4.2.3 項に示した副作用を解消する必要がある. 細分化モデルの操作は全て同時代入であるため代入順序を考慮せずに結合することができるが, 細分化実装の操作は代入順序を持つため, 結合順序によって得られる結果が異なる. これを解消するため, 4.2.3 項の細分化実装は代入文を ‘大域変数を参照して計算結果を局所変数に格納する参照部’ と, ‘その局所変数を参照して大域変数への代入を行う代入部’ に分割して持つ. これにより操作結合時に全ての参照部が代入部の前に実行されるよう結合することで副作用を解消している.

8.4 節の MSSS の適用例ではマイレージ付きレンタカーシステムを合成した. この例では ‘要求モデルへの初期化の追加’ と ‘部品が割り当てられなかった細分化モデルに対する部品追加’ を行うことで合成ソフトウェアの証明責務が全て真になった. 副作用を解消した合成例を図 8.22 に示したが, この合成結果の副作用が解消されていることは証明責務が全て真であることからも明らかである. ただし, 図 8.22 の合成結果において同じような WHILE ループや IF 文による判定を複数回行っている事からも分かるように, 提案手法によって得られる合成ソフトウェアは信頼性は高いが実行効率は低い. これは, 本研究の関心事が合成ソフトウェアの信頼性であり, 実行効率の向上のために手法が複雑化する事を避けたためである. このため, 6.4.2 項で与えた実装操作の合成手順は 4.2.3 項の部品の定義で与えた副作用の解消に忠実である. また, 実行効率の問題は合成ソフトウェアにアルゴリズムの最適化を適用することで解消できると考える.

9.2.2　MSFC 生成の信頼性

4.3 節では B Method の枠組みに基づいて部品の信頼性を定義した. この様に信頼性を証明責務という数学的な命題に基づいて定義することで手法自体の信頼性を定義に基づいて評価することができる. 本節では提案した MSFC 生成の信頼性を部品の信頼性に基づいて評価する.

モデル細分化の信頼性

4.3 節では B Method の証明責務に基づき細分化モデルの証明責務を定義した．さらに, 5.7 節ではモデル細分化で得られる細分化モデルの証明責務が常に真であることの証明を試みた．この結果, 初期化から得られる細分化モデルの証明責務は常に真になることが証明された．また, 制約条件展開が任意の変数間の暗黙的な関係を全て推論できるならば操作から得られる細分化モデルの証明責務は真になることが証明された．この ‘制約条件展開が任意の変数間の暗黙的な関係を全て推論できるならば’ という仮定は証明 2 において理想的な推論器を仮定した事で生じる．しかし，現実には理想的な推論器は定理証明と同様に停止性を保証出来ないため提案したモデル細分化手法では 5.3 節で示したように有限個のルールを有限回適用することで近似的に推論を行っている．このため, モデル細分化では得られた細分化モデルに対して証明責務の証明を行う．細分化モデルの証明責務は B Method のモデルの証明責務と同じであるため, この作業には AtelierB[6] などの証明支援環境が利用でき，自動証明器で証明できない証明責務のみを人手により証明する．

実装抽出の信頼性

実装抽出の信頼性は ‘細分化実装と実装依存モデルの証明責務が真になること’ と定義できる．MSFC 生成の実装抽出では 7.2.6 項に示した証明責務最小化により証明責務が真であることを保つよう証明責務を最小化する．このため, 実装抽出の信頼性は原理的に保証されている．ただし, 証明責務最小化は 7.2.6 項に示したように定理証明を完全に自動化することが出来ないために作業者の負担軽減が問題となる．MSSS フレームワークではこの問題を部品の細粒度化により解消している．仮定削減作業の定理証明は経験的に証明責務のゴールが大きい程難しい．このゴールの大きさは仮の不変条件と操作抽出で得られた操作の事前条件で決まる．そのため, 操作抽出で得られる操作が小さく, その操作が含む大域変数が少ない程, 仮定削減の定理証明は容易になる．本手法では部品の粒度を小さくしたことで部品群が大量に生成されるが, 仮定削減の定理証明が容易であるため自動定理証明が期待できる．自動定理証明が失敗した部品に対しては作業者が証明を行うが, この証明作業も 7.2.6 項に示したように細分化元ソフトウェアで証明を行った者が証明を行うことで作業負担を軽減できる．証明の作業量と証明責務が偽である場合の作業量については 9.4 節で考察する．また，5.7.2 項の証明 2 では制約条件展開の推論能力が高いほどモデル細分化の信頼性が向上することを示した．このため, 細分化モデルの証明責務が偽になる場合には制約条件展開を修正することでモデル細分化手法の信頼性が向上する．

9.3　計算量

定理証明に基づく自動コード生成手法は信頼性は高いが定理証明の時間的計算量も大きく実用化が困難であった．MSSS フレームワークもソフトウェアの合成と信頼性保証に定理証明を応用するため，以下の工夫により時間的計算量を軽減している．

1. モデルと実装間の整合性は MSFC における実装依存モデルと細分化実装の整合性から保証する．
2. 部品検索は細分化モデルに対して行い, 実装を直接検索しない．
3. 部品リポジトリにおいて部品は細分化モデル毎に格納され，細分化モデルは階層構造を持つ．
4. モデル標準化により細分化モデル間の数学的判定を文字列一致で行う．

工夫 (1) は計算量に対する信頼性の対費用効果を向上する．この対費用効果の向上は既存のソフトウェア部品再利用においても ‘再利用した機能の単体テストが不要’ などの形で得られていた．これに対して，MSSS フレームワークではソフトウェアの信頼性を動作テストではなく定理証明により評価するため，MSFC の信頼性を保証することでソフトウェア合成時の定理証明に必要な手間と計算量が軽減される．

工夫 (2) では要求モデルを部品と同じ粒度に細分化した検索キーを用いて部品の仕様を検索することで判定に要する計算量と判定回数を低減する．一般的に実行可能コードと数学的仕様は文字列として決して一致しないため, 実行可能コードが数学的仕様を満たすか否かは定理証明を用いなければ判定できない．これに対して MSSS は同じ粒度の数学的仕様同士を比較するため, 数学的仕様の表現を統一することで出来れば文字列一致による判定が可能となる．MSSS ではモデル標準化により細分化モデルの表現を統一している．

工夫 (3) では細分化モデルを部品に対するダイジェストとして部品を検索することで判定回数を低減する．細分化モデルは部品の仕様であるため，同じ仕様であっても実装方法などの違いにより複数の部品が存在し得る．このため，細分化モデルが部品に対するダイジェストとして機能し，部品の判定回数が低減される．ただし, 検索で得られた部品間で実装方法が矛盾する可能性があるため, 部品選択で無矛盾判定と実装方法の選択を行う．また，4.4.2 項のように部品リポジトリの細分化モデル群を階層化することで 6.2.2 項に示した部品検索時の探索空間の限定が可能になる．これにより部品検索時の判定回数が低減される．

工夫 (4) は大量の部品が格納されたリポジトリにおける部品検索の計算量を低減する．この仕組みを図 9.1 に示す．単純に形式仕様間の数学的判定で部品検索を行うと図 9.1 の右側のように部品数だけ定理証明を行う必要があり，現実的な時間で部品を得られない．これに対して MSSS では 5.3 節の制約条件展開と 5.6 節の構文要素整列により字面を統一することで検索キーと部品の仕様との間の判定を文字列一致で行う．この時，部品数は

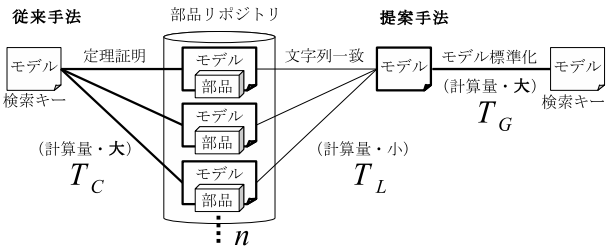


図 9.1: 文字列一致判定による計算量低減

膨大であるため全体の計算量を低減することができる．計算量を簡単に試算すると以下のようになる．要求仕様を細分化した細分化モデルと部品の細分化モデル間の数学的判定の計算量を T_C, モデル標準化の計算量を T_G, 標準化された細分化モデル同士の字面一致の計算量を T_L, リポジトリに格納された部品数 n とする．この時，検索キー 1 つあたりの部品検索の計算量は工夫 (4) を適用しない時に nT_C であるのに対して, 工夫 (4) を適用することで $T_G + nT_L$ となる．ここで, T_C と T_S は伴に定理証明を含むため $T_L \ll T_C$ となり, n が大きいほど工夫 (4) による計算量低減効果が大きい．モデル細分化は暗黙的な命題を制約条件展開で推論するため, T_S は定理証明と同等の計算量を必要とする．制約条件展開は元々は細分化モデルの無矛盾性を保証するための処理だが, 本手法では ‘数学的に等しいモデルにおける出現する命題の統一’ にも用いる．これはモデル標準化による ‘命題の並びを統一’ により数学的に等価な命題の字面を一致させるために必要となる．5.3 節で提案した制約条件展開における推論は停止性を保証できないため推論は近似的にならざるを得ない．このため，制約条件展開の展開ルールを整備する際には検索の完全性と計算量のトレードオフを考慮する必要がある．

9.4　作業量

図 9.2 は MSSS フレームワークにおいて人間が行う作業を表した図である．この様に, MSSS と MSFC 生成では以下の作業を人間が行う．

- MSSS
  1. 要求モデルの記述
  2. 細分化モデルの検証，修正
  3. 部品選択
  4. 合成ソフトウェアの検証，修正

- MSFC 生成
  1. 細分化モデルの検証，修正
  2. 仮定削減

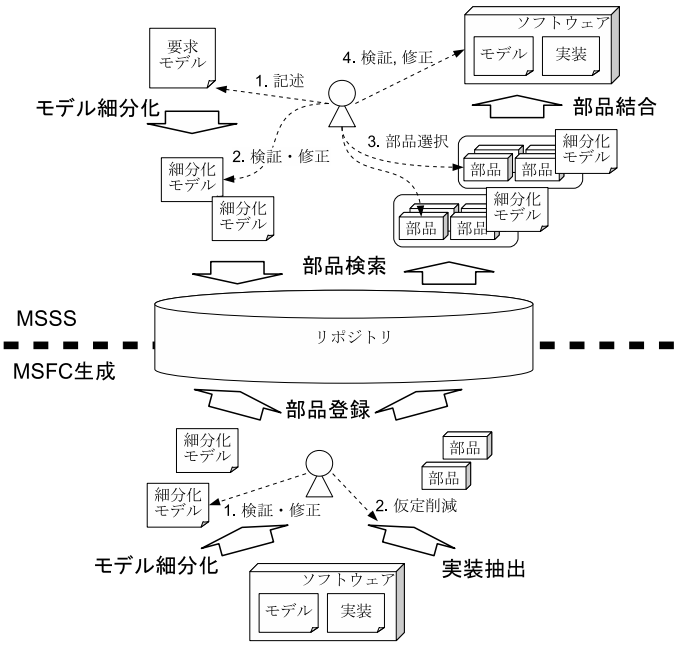


図 9.2: MSSS フレームワークにおける人間の作業

以下の節ではこれらの作業と，その低減手法を考察する．

9.4.1　要求モデル記述のコスト

利用者が記述する要求モデルは実装非依存なモデル変数のみに注目した B Method のモデルである．B Method では‘モデルの事前条件が真ならば実装の事前条件も真である’という制約があるため，‘ファイルが開いていること’のようなモデルの代入には現れない実装依存の制約条件を，実装だけでなくモデルの事前条件に記述する必要がある．

これに対して，MSSS では細分化モデルの代入中に現れない実装依存の変数とその制約条件を実装依存モデルに集約する．これにより，MSSS では実装非依存な細分化モデルを用いた実装方法に依存しない部品再利用を可能にしている．このため，MSSS の要求モデルを記述する際には実装方法を考慮する必要がなく，一般的な B Method のモデル記述より容易である．なお，実装方法の指定は部品選択において対話的に行う．

9.4.2　細分化モデルの検証，修正のコスト

細分化モデルの検証

細分化モデルは MSFC の仕様であり，また，MSSS では MSFC に対する検索キーでもある．この細分化モデルの信頼性は 4.3 節で示したように，B Method のモデルの証明責

務を元に数学的に定義される．MSFC 生成と MSSS では共通のモデル細分化手順によりモデルから細分化モデル群を生成する．本手法ではこのモデル細分化手法自体の信頼性を数学的に保証することで，細分化モデルの検証と修正に要する作業量を低減することを試みた．

9.2.2 項の ‘モデル細分化の信頼性’ では入力モデルが無矛盾である場合に細分化モデルが無矛盾となる事を定理証明により保証することを試みた．この結果，細分化対象が初期化である場合には得られる細分化モデルが必ず無矛盾であることを保証できた．しかし，モデル変数を代入中で参照するような操作については矛盾が生じ得ることが証明された．このため，この様な細分化モデルに対しては細分化後に証明責務の証明を行い，証明責務が偽である場合には真になるよう修正する必要がある．

この細分化モデルの証明責務の証明は証明支援環境を利用すると共に，細分化元ソフトウェアでの証明作業者が行うことで作業負担を軽減できる．証明支援環境には AtelierB などの B Method の証明支援環境が利用できる．これは細分化モデルの書式を B Method のモデルに基づいて定めており，変数の初期化を追加するだけで B Method のモデルとして扱えるためである．これにより，細分化モデルの検証では自動証明できない証明責務のみを人手により証明すれば良い．また，細分化モデルの証明責務は細分化元ソフトウェアの証明責務のゴールと仮定を小さくしたものである．このため，細分化モデルの証明責務は細分化元ソフトウェアに現れる証明責務と良く似たものになり，細分化元で証明作業をした作業者がこれを行うことで作業負担を軽減できる．

細分化モデルの修正

細分化モデルの修正も細分化元の証明作業者が行うことで作業負担を軽減できる．細分化モデルの修正は具体的には 5.3 の制約条件展開で推論できなかった変数間関係を制約条件に追加することである．9.2.2 項の定理証明によるモデル細分化の信頼性保証では，変数 x, y 間の暗黙的な変数間関係 I_{xy} が不変条件に存在し，細分化元の操作で変数 x が変数 y を参照して変化し，また，同一操作中で y が変化する際，変数 x についての細分化モデルの証明責務が偽になりうる事が示された．この原因は制約条件展開における推論が変数間関係 I_{xy} を導出できる保証が無いためである．しかし，細分化元ソフトウェアでも証明責務を証明するにはこの変数間関係 I_{xy} を導出する必要があるため，細分化元モデルの証明作業者にはこの変数間関係の追加が容易である．

また, 証明 2 では制約条件展開の推論能力が高い程, 信頼性も向上することが示された. このため, 制約条件展開を修正することでモデル細分化手法の信頼性が向上し，作業者の負担を軽減できる．8 章では銀行口座システム，グループ管理システム，マイレージ付きレンタカーシステムのモデルに対してモデル細分化を適用したが，生成された細分化モデルは全て証明責務が真であった．この例題では 5.3 節で示した制約条件展開の推論ルー

ルを用いているが，この推論ルールの数は 20 個程度である事から，ルール整備によるモデル細分化手法の信頼性の向上は難しくないと考えられる．

9.4.3　部品選択のコスト

MSSS では MSFC 生成により B Method で構築された既存ソフトウェアから部品を自動生成できる．これは部品整備を容易にし，自動生成の適用可能性を向上するのに有効であるが，一方で，大量の部品が生じるために部品把握が困難になる．例えば，‘あるモデル変数を+1 する部品’，‘+2 する部品’, . . . のように粒度の細かい部品や，特定の場面でのみ利用できる再利用性の低い部品が無数に発生する．また，Java の標準 API では 3700 以上のクラスが存在するが，提案手法で扱う MSFC の粒度はさらに細かいため，部品数はそれ以上に膨大になると考えられる．

一般的に部品を大量に用意できる手法ほど，部品を把握するのにコストがかかり，逆に部品が少ないと合成手法の適用性が低下する．この様に，部品整備コストと部品把握コストはトレードオフの関係にあり，MSFC のように部品自動生成で部品整備コストを低減する場合，部品把握コストへの対応が必要となる．提案手法ではを手法全体を通して部品の読解を不要にし，利用者から部品を隠蔽することでトレードオフの問題を解決している．

部品検索では要求モデルからの検索キーの自動生成と数学的判定による部品検索により利用者から部品を隠蔽している．例えば，利用者が ‘モデル変数を+1 する’ などの要求を要求モデルに記述すると，要求モデルから検索キーが自動生成され，モデル変数を+1 する部品が得られる．一般的にクラスや関数などの再利用では型判定や自然言語により仕様を検索するが，この場合，作業者が仕様を読解して挙動を理解する必要がある．これに対して提案手法では要求仕様に挙動を記述し，6.2 項のように人手によらず部品を検索するため，部品検索に関して部品数増加による人的コストの増加は生じない．

部品選択では部品の読解が不要な実装方法の選択 (6.3.3 項) を提案することで利用者から部品を隠蔽している．この部品選択はモデル変数の実装に用いるモジュールを選択することで行う．このため，提案手法では 3.3.1 項において ‘MSFC で利用するモジュールは単一の概念に対する操作を集約する’ と制約を与えた．この様に操作を集約することで 3.3.1 項で分類したように入出力と永続化のモジュールは十分に少なくなると考えられる．また，‘ファイル’ や ‘制御対象の機械’ などの概念ごとにモジュールを構築することで ‘モデル変数が表す概念’ という観点からモデル変数に対して容易にモジュールを選択できる．また，一般的なライブラリではキューとスタックが別部品となる場合があるが，FIFO と FILO の様な数学的に表現可能な違いはモジュールではなく，細分化モデルで解消されるため，3.3.2 項で分類したように入出力と永続化のモジュールは十分に少なくなると考えられる．

なお，合成したソフトウェアの証明責務が偽になり部品を選択しなおす場合には 6.5 節で述べたように同じ矛盾が生じる部品の組み合わせを排除できる．これは証明責務の証明時に矛盾の原因となる命題が明らかになり，部品選択時にその命題を持つ部品群を除外すればよいためである．ただし，提案手法は型のみで判定するため精度が低く，新たな矛盾が生じる可能性がある．この解決として証明責務が偽になることが明らかな組み合わせを計算時間を限定した自動定理証明により排除することが考えられる．この場合，証明責務の否定を証明することや，膨大な部品の組み合わせに対する計算時間の考慮が必要になる．

1 つのモデル変数に 2 つ以上のモジュールを選択したい場合は，要求モデルの詳細化してモデル変数を分割する事で対応できる．例えば，データベースで永続化するモデル変数 x_a を特定の操作においてファイルに出力する事が考えられる．この場合は詳細化により $x_b = x_a$ という不変条件を持つモデル変数 x_b を追加し，x_b にファイルを扱うモジュールを選択する．集合論で I/O を表現するにはモデル変数として入力元や出力先の集合や順序列を明示し，その集合間の要素移動を画面出力やファイルへの書き込みと捉える．このため，各モジュールごとにモデル変数を指定することは妥当だと考える．

提案した部品選択手法の限界として整列方法などのアルゴリズムを指定できない点が挙げられる．これは検索キーとなる細分化モデルでは計算結果のみを表現するためアルゴリズムを区別できず，また，モジュールは概念ごとに構築するため，モジュール選択では汎用的なアルゴリズムを指定できないために生じる．一方で，この問題は作業者は計算結果に影響しない些末なアルゴリズムを考慮しなくて良いという利点も生じる．

9.4.4　生成ソフトウェアの検証と修正コスト

検証と修正

合成ソフトウェアの検証では出力モデルと出力実装に対して B Method の枠組みで検証を行う. 本手法では利用する高信頼ソフトウェア部品は実装依存モデルと細分化実装間の整合性が保証されるため, 部品間の不整合のみに注目して検証すれば良く，証明作業を軽減できる. 合成ソフトウェアが不完全である時には部品の追加などが必要であるが, 6.5 節に示した補助により合成されたコードを読解せずに部品を追加できる. 6.5 節の例では細分化モデル RentC5 に対して部品が割り当てられないが，類似部品として部品 C_{G3} が提示されるためコードを記述すること無く実装が得られた.

B Method におけるソフトウェア合成手法 として BART[7] が挙げられる．BART は B Method の断片的なモデルと実装の組を部品として持ち，部品のモデルと仕様をパターンマッチすることで実装の雛形を得る．BART が本手法と異なる点として ‘部品の信頼性を保証しない’，‘部品の粒度を定めていない’ ことが挙げられる．MSFC では部品の信頼性

を保証できるため，部品間の矛盾のみに注目して合成ソフトウェアの証明作業を軽減できる．また，MSSS が不足部品の仕様を提示できる理由は部品の粒度が固定であり，要求モデルから部品の仕様である細分化モデルを一意に生成できるためである．一方で BART のように部品の粒度を定めない手法は追加すべき部品の仕様を一意に定められないため，この様な補助が困難である．

新規部品の開発

ソフトウェアの修正では新規部品の開発が必要になる場合がある．2.2.2 項で紹介したソフトウェアプロダクトライン (SPL) では部品開発者と部品組立者を分け，開発組織を分業化することでそれぞれの独立性と専門性を高めている．この方法ではコードの二重開発の防止や部品の信頼性の向上が見込まれる．一方で管理すべき部品が膨大になると人手による部品の管理が困難になる．特に MSSS のように細粒度部品リポジトリと呼ばれる単純な機能の部品が大量に登録される部品リポジトリでは，部品へのタグ付け等による部品検索ツール [28] などの計算機による開発補助 (CASE ツール) の導入が前提となる．提案手法では要求モデルからの部品検索をモデル細分化により自動化し，二重開発を防止している．また，不足部品の形式仕様は要求モデルから MSSS が自動生成するため，部品開発者への部品開発依頼が容易であり，自然言語と異なり仕様伝達時の齟齬や解釈の違いを防止できる．これは特に，部品の開発を海外にオフショアする場合に有効であると考えられる．一般的に，第三者に製品開発を依頼する際には最終的に要求を満たした製品であることを受入テストにより判断するが，オフショア開発では双方で記述/解釈可能な自然言語で仕様等の知識の伝達を行うため，仕様の翻訳時のミス，母国語以外での知識伝達のミス等が発生し，受入テストの段階で問題が顕在化する場合がある．この問題に対して UML など，双方が共通に持つ知識 (UML ならばオブジェクト指向) を基盤とした仕様記述言語を用いることの有効性が報告されている [33, 38]．MSSS 手法でも知識伝達に数学を基盤とした形式仕様を用いることで翻訳時や知識伝達時のミスを抑止する．また，B Method の枠組みで部品開発を行うため，受入テスト前に開発サイドで十分なテストを行う事が可能である．この様に，MSSS 手法を SPL の枠組みで利用することで SPL の問題点を改善できる．また，MSSS 手法だけでは UI 開発ができないため，MSSS を SPL の枠組み内で利用し，MSSS が生成した機能群と UI を部品組立者が結合することが考えられる．

9.4.5　仮定削減コスト

仮定削減は実装抽出において与えられた細分化モデルに対して細分化実装の証明責務が真になることを保証するための処理である．7.2.6 項で述べたように仮定削減は機械証

明の補助を受けて式 (7.6) に示した命題を定理証明し，この過程で証明に寄与しない仮定を削減することで行う．このため，単純に仮定削減を行うと定理証明と同様の作業量を要する．この定理証明は完全に自動化できないため，作業者の負担軽減が問題となる．

経験的に仮定削減作業の定理証明は証明責務のゴールが大きい程難しくなる．このゴールの大きさは仮の不変条件と操作抽出で得られた操作の事前条件で決まる．そのため, 操作抽出で得られる操作が小さく, その操作が含む大域変数が少ない程, 仮定削減の定理証明は容易になる．本手法では部品を粒度化したため，大量の部品群が生成されるが, それぞれの仮定削減の定理証明は容易になるため，多くの部品には自動定理証明が期待できる．

一方で，要求モデルが複雑な条件を持つ場合や実装で WHILE ループを利用する場合には自動定理証明が期待できず，人間による証明が必要となる．この場合でも，MSFC の証明責務は細分化元ソフトウェアの証明責務のゴールを小さくした命題であるため，細分化元ソフトウェアで証明を行った作業者であれば容易に証明できる．

以上のことから細分化元ソフトウェアの証明責務を証明した者が仮定削減を行うことで，作業者負担を低減できる．特に，仮定削減により実装抽出の信頼性が原理的に保証されることを考慮すると，その作業者負担は手法の信頼性に対して十分に小さいと言える．

第10章

おわりに

本論文では既存のソフトウェア自動合成の課題である‘生成ソフトウェアの信頼性向上’と‘手法適用に要するコストの低減’を目的とし，モデル充足ソフトウェア合成 (MSSS) フレームワークを提案した．MSSS フレームワークは形式手法による高信頼ソフトウェア開発を部品再利用により自動化するフレームワークであり，その形式手法には仕様と実装間の整合性を保証できる B Method を応用した．また，部品として B Method を応用したモデル充足細粒度部品 (MSFC) を定義し，部品の信頼性を静的に保証可能にした．既存の部品再利用によるソフトウェア合成では各ソフトウェアドメイン毎に高信頼なソフトウェア部品を整備する必要があり，MSFC のような高信頼なソフトウェアを整備することが手法適用を阻害するコスト要因となっている．これに対して MSSS フレームワークでは B Method で構築された既存ソフトウェアを細分化し，信頼性が保証された MSFC を得る MSFC 生成を提案した．

MSSS フレームワークの新規性としては以下が挙げられる．

1. 機械的に再利用することを前提に細粒度なソフトウェア部品を自動生成する点
2. 部品自動生成時に部品の静的な信頼性を保証する点
3. 部品再利用に定理証明に基づく健全かつ完全性の高い部品検索を利用する点
4. ソフトウェア合成時に不足部品の仕様が提示され，これを追加することで部品リポジトリが成長する点

これらの新規性を実現するために，MSSS フレームワークではソフトウェア部品の粒度を定義し，モデルと実装をその粒度に一意に細粒度化する手法を提案した．さらに，その細粒度化手法の信頼性を定理証明に基づき保証した．この様な部品は部品数が膨大になるため，部品再利用時の検索コスト低減と人手に依らない部品再利用が必要となる．検索コスト低減にはモデルの字面統一による数学的等価性の文字列比較による判定を提案した．これにより，高い健全性と完全性が求められる自動部品再利用に自動生成で得られた膨大な部品群を活用できる．

また，数学的に表現困難な実装方法に依存する要求に合致する部品を膨大な部品群か

ら選択するため，MSSS フレームワークでは部品を読解すること無く要求モデルと輸入モジュールの知識のみで適切な部品を選択できる部品選択手法を提案した．従来のソフトウェア合成で手法適用性の障害となっていた部品の整備性については，MSFC 生成が部品が未整備な問題領域における手法適用性を向上すると共に，ソフトウェア合成時の不足部品の仕様提示により部品リポジトリの成長が容易に行える．この様に不足部品の仕様を提示できるのは細分化モデルの粒度が一意に定まっており，細分化された検索キーがそのまま部品の仕様となるためである．

この様に MSSS フレームワークは多くの優れた点を有する一方で，以下のような課題も挙げられる．

1. 制約条件展開に用いられる推論規則が研究で用いた構文要素にしか対応していない．
2. 部品の選択可否判定が単純な型判定のみで行われるため，健全性が低く，部品合成後に部品選択のやり直しが生じる場合がある．
3. MSSS フレームワークが基盤とする形式手法 B Method の実用例が国内で公開されておらず，例題とそこから得られる知見が不十分である．

特に，国内での形式手法の実用例不足については，形式手法の研究と啓発を続けることで，産学連携による形式手法の実用を促進することが重要であると考える．形式手法が対象とする問題領域は現状では人命に関わる交通や機械制御に限定されており，企業活動のなかでの形式手法の応用はまだまだ不十分である．その要因は形式手法のコストの高さと，それを扱う人材の不足にあると考えられる．MSSS フレームワークは形式手法による高信頼ソフトウェア開発を自動化するものであり，形式手法の適用コスト削減と必要な人材の問題を解決する可能性を持つ．

現在でも実装作業がルーチンワークに準ずることは十分な認知されているが，それがどこまで自動化できるかについての知見はまだまだ不十分であると考えられる．MSSS フレームワークはリポジトリが十分に成長すれば要求モデルの記述のみで高信頼ソフトウェアが生成できることを示唆している．今後もソフトウェア開発の自動化と高信頼化に貢献し，人々が企画や GUI デザイン，ユーザ体験の思案など，より創造的な仕事に専念できる未来を模索したい．最後に，高信頼ソフトウェア開発の自動化はソフトウェア工学における長年の夢であり，本研究もその到達への一歩へ貢献できれば幸である．

謝辞

本論文は著者が 電気通信大学 博士前期/後期課程 の在学中に研究，投稿した論文を単位取得退学後に学位論文としてまとめたものであります．長期に渡る研究のため，この間に多くの方々にお世話になりました．ここではお世話になった皆さまへ一言お礼の言葉を述べさせて頂きたいと思います．

まずはじめに，終始，本研究の指導的立場にありました織田 健 助教に心から感謝致します．織田先生には学部時代のリテラシー教育からお世話になっておりますが，特に，博士前期課程からは他研究室から渡辺研究室に移って来た私に対し，形式手法の何たるか，再利用の必要性をご教授頂き，また，議論を通して数多くの知見を頂きました．日々の生活におきましても，人生の先達として様々な見識や人生観，価値観への示唆を頂き，挫けそうな折には酒杯を重ねて激励して頂きました．織田先生には感謝の言葉が尽きません．

次に渡辺研究室にて指導教員としてご指導下さった渡辺 成良 名誉教授に感謝致します．渡辺先生にはソフトウェア工学だけでなく，マルチエージェントシステムを通したモデル化と分析，教育工学を通した人と人，人と計算機の結び付きの可能性について，様々な知見や価値観を与えて頂きました．また，ご退官後も事ある毎に大学に足をお運び頂き，研究への助言と激励を頂けたことは大変感謝しております．渡辺先生がご退官後に指導教員を引き受けてくださいました西野哲朗教授には論文執筆におきまして様々な助言を頂きました．特に審査委員会においては主査として暖かく見守って頂き，また，委員の編成や公聴会の開催など，様々な面でご尽力頂き，大変，感謝しております．

渡辺研究室，織田研究室，西野研究室の皆さまには議論や日々の生活を通して大変お世話になりました．渡辺先生がご退官後，一時，研究室の所属が私一人となり，研究室に一人篭る時期がありました．この時期を通して研究室の仲間が如何に研究面，精神面で支えであったかを痛感致しました．特に渡辺研究室，織田研究室のソフトウェア工学研究グループの先輩，同輩，後輩の皆さまには大変お世話になりました．

5.7 節の証明による手法の信頼性保証を考えるにあたっては，学部の卒業研究時代に垂井研究室で証明問題に取り組んでいたことが大きく寄与しております．垂井研究室にてご教授下さいました 垂井 淳 順教授には今一度お礼の言葉を述べさせて頂きます．また，本論文を纏めるに当たり，博士論文審査委員会の西野哲朗先生，渡辺成良先生，高橋治

久先生，柏原昭博先生，寺田実先生，庄野逸先生にはご多忙の中，この様な長編の論文を熟読して頂けましたことを本当に感謝しております．

以上の様に，本論文は著者だけでなく，議論に参加して頂いた皆さま，そして参考文献において素晴らしい成果と知見を残してくださった先達方があって始めて成ったものと感じております．誠にありがとうございました．

研究以外におきましても，学科計算機室でお世話になりました高木一幸先生，技官の服部先生，TA や様々な手続きでお世話になった学科事務室の皆さま，忙しい中で様々な融通を頂いた現在の所属先である株式会社シフトの皆さま，学資面で並々ならぬご助力を頂いた学生課の皆さま，日本奨学金機構様，昭和池田記念財団様，そして，ここまで暖かく見守って下さった家族に，この場を借りて深く感謝を申し上げます．

関連論文の印刷公表の方法及び時期

査読論文

1. 中村丈洋, 織田健, 西野哲朗
   論文題目 “B Method における高信頼ソフトウェア部品自動生成”
   2011 年 11 月情報処理学会 論文誌, Vol.52, No.11, pp.2989-3007
   (4 章，5 章，7 章 に関連)
2. 中村丈洋, 織田健, 西野哲朗
   論文題目 “高信頼細粒度部品再利用による形式手法におけるソフトウェア合成”
   2013 年 8 月情報処理学会 論文誌, Vol.54, No.8, pp.2012-2024
   (5 章，6 章 に関連)

発表論文

1. 中村丈洋, 織田健
   “再利用による自動コード生成を目的とした B Method におけるソフトウェアの部品化”
   情報処理学会研究報告, vol. SE163, no. 31, pp. 169-176, 2009
   (5 章，7 章 に関連)
2. 中村丈洋, 織田健
   “B Method における部品再利用による自動コード生成のための仕様細分化”
   ディペンダブルシステムシンポジウム, DSS2009, pp. 159-168, 2009
   (5 章 に関連)
3. 中村丈洋, 織田健
   “B Method における自動コード合成フレームワークの提案”
   情報処理学会研究報告, vol. SE170, no. 18, pp. 1-8, 2010
   (3 章 に関連)
4. 中村丈洋, 織田健
   “形式仕様を用いた部品検索における計算量低減”

情報処理学会研究報告, vol. SE173, no. 3, pp. 1-8, 2010
(5 章 に関連)

5. Takehiro Nakamura, Takshi Oda, Tetsuro Nishino
“Model Satisfiable Software Syntehsis Method for Formal Development”
International Workshop on Modern Science and Technology 2012 (IWMST2011), Tokyo, Japan, August 30-31, 2012.
(3 章 に関連)

参考文献

[1] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: the KobrA approach. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pp. 289–310, 2000.

[3] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. Fischer Nilsson. Synthesis of programs in computational logic. *LNCS*, Vol. 3049, pp. 30–65, 2004.

[4] S. Blazy, F. Gervais, and R. Laleau. Reuse of specification patterns with the B method. In *ZB 2003: Formal Specification and Development in Z and B*, pp. 626–626. Springer, 2003.

[5] CEA. Papyrus uml. http://www.papyrusuml.org/. accessed 2010-12.

[6] CLEARSY. Atelier B. http://www.atelierb.eu. accessed 2010-07.

[7] CLEARSY. BART: B automatic refinement tool. http://www.tools.clearsy.com. accessed 2010-07.

[8] CLEARSY. ComenC: B0 implementation translation into c language. http://www.comenc.eu/. accessed 2010-07.

[9] David Dikel, David Kane, Steve Ornburn, William Loftus, and Jim Wilson. Applying software product-line architecture. *Computer*, Vol. 30, No. 8, pp. 49–55, 1997.

[10] Tomaž Dogša and David Batič. The effectiveness of test-driven development: an industrial case study. *Software Quality Journal*, Vol. 19, No. 4, pp. 643–661, 2011.

[11] G.C. Gannod, Y. Chen, and B.H.C. Cheng. An automated approach for supporting software reuse via reverse engineering. In *13th IEEE International Conference on Automated Software Engineering, 1998. Proceedings*, pp. 94–103, 1998.

[12] David Garlan and Dewayne E Perry. Introduction to the special issue on software architecture. *IEEE Trans. Software Eng.*, Vol. 21, No. 4, pp. 269–274, 1995.

[13] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2003.

[14] J. Jeng and B. Cheng. Using formal methods to construct a software component library. *LNCS 717*, Vol. 717, pp. 397–417, 1993.

[15] Z. Manna and R.J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, Vol. 14, No. 3, pp. 151–165, 1971.

[16] B. Meyer. The grand challenge of trusted components. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 660–667. IEEE, 2003.

[17] Jenkins Org. Jenkins: An extendable open source continuous integration server. http://jenkins-ci.org/.

[18] JUnit Org. Junit: A programmer-oriented testing framework for java. http://junit.org/.

[19] Massimo Paolucci, Takahiro Kawamura, Terry Payne, and Katia Sycara. Semantic matching of web services capabilities. In *The Semantic Web ISWC 2002*, pp. 333–347. Springer, 2002.

[20] F. Plasil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, pp. 43–51. IEEE, 1998.

[21] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pp. 43–54. Springer, 2005.

[22] D.R. Smith. KIDS: A semiautomatic program development system. *Software Engineering, IEEE Transactions on*, Vol. 16, No. 9, pp. 1024–1043, 1990.

[23] R. Van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, Vol. 33, No. 3, pp. 78–85, 2000.

[24] M. Weiser. Program slicing. In *ICSE*, pp. 439–449, 1981.

[25] 吉田一, 山本晋一郎, 阿草清滋. XML を用いた汎用的な細粒度ソフトウェアリポジトリの実装. 情報処理学会論文誌, Vol. 44, No. 6, pp. 1509–1516, 2003.

[26] 橋本靖, 山本晋一郎, 阿草清滋. Program slicing を利用したプログラムカスタマイザ. 電子情報通信学会技術研究報告, Vol. SS94, No. 10, pp. 73–80, 1994.

[27] 玉井哲雄. ソフトウェア工学の現状と動向：ソフトウェア開発への知識工学の応用. 情報処理, Vol. 28, No. 7, pp. 898–905, 1987.

[28] 古山将佳寿, 山本晋一郎, 阿草清滋. ドキュメントを含むソフトウェアモデルの提案. 日本ソフトウェア科学会 FOSE, Vol. 99, pp. 100–107, 1999.

[29] 戸板晃一, 山本晋一郎, 阿草清滋. XML を用いたソフトウェア関連文書とソースプログラムの整合性検査ツール. 日本ソフトウェア科学会 FOSE2001, pp. 129–140, 2001.

[30] 二木厚吉, 外山芳人. 項書き換え型計算モデルとその応用. 情報処理, Vol. 24, No. 2, pp. 133–146, 1983.

[31] 三鍋孝介. B Method におけるモデル細分化アルゴリズム. 卒業研究論文, 電気通信大学, 2012.

[32] 独立行政法人情報処理推進機構. 形式手法適用調査. http://www.ipa.go.jp/files/000004548.pdf.

[33] 中原俊政, 藤野博之. オフショアプロジェクトにおける UML の適用. プロジェクトマネジメント学会誌, Vol. 10, No. 6, pp. 9–14, 2008.

[34] 井田哲雄. 計算モデルの基礎理論. 岩波書店, 1991.

[35] 福安直樹, 山本晋一郎, 阿草清滋. 細粒度リポジトリに基づいた CASE ツール・プラットフォーム Sapid. 情報処理学会論文誌, Vol. 39, No. 6, pp. 1990–1998, 1998.

[36] 峰岸巧, 永田守男, 神谷慎吾, 山本修一郎, 安東孝信, 山城明宏. MDA に基づくソフトウェア開発の事例と開発プロセス. Technical Report 22(2002-SE-140), 慶應義塾大学, 慶應義塾大学, 株式会社　NTT データ, 株式会社　NTT データ, 株式会社　東芝, 株式会社　東芝, 2003.

[37] 外山芳人. 完備化による等式証明. 人工知能学会誌, Vol. 16, No. 5, pp. 668–674, 2001.

[38] 木崎悟, 成田亮, 丸山英通, 中鉢欣秀. グローバルなソフトウェア開発におけるマネジメント手法. 情報処理学会研究報告. ソフトウェア工学研究会報告, Vol. 2011, No. 1, pp. 1–8, 2011.

[39] 鷲崎弘宜, 深澤良彰. ソフトウェアパターン研究の現在と未来. Technical Report 55(2003-SE-141), 早稲田大学理工学部, 早稲田大学理工学部, 2003.

付 録A

書き換えルール群

A.1 プリミティブ化の書き換え規則

書き換え元	書き換え後
$a \neq b$	$\neg(a = b)$
$a \notin b$	$\neg(a \in b)$
$a \subseteq b$	$a \in \mathbb{P}(b)$
$a < b$	$a \leqslant b \wedge \neg(a = b)$
$a \geqslant b$	$b \leqslant a$
$a \nrightarrow b$	$\{r \mid r \in a \leftrightarrow b \wedge \mathrm{dom}(r) \subseteq a \wedge \mathrm{ran}(r) \subseteq b \wedge (r^{-1};\ r) \subseteq \mathrm{id}(b)\}$
$a \rightarrow b$	$\{r \mid r \in a \nrightarrow b \wedge \mathrm{dom}(r) = a\}$
$a \leftrightarrow b$	$\mathbb{P}(a \times b)$
$\mathrm{ran}(r)$	$\mathrm{dom}(r^{-1})$
$u \lhd r$	$\mathrm{id}(u);\ r$
$u \lhd\!\!\!- r$	$(\mathrm{dom}(r) - u) \lhd r$
$q < +r$	$(\mathrm{dom}(r) \lhd\!\!\!- q) \cup r$

A.2　推論規則

規則		
$X \in \mathbb{P}(Y) \land Y \in \mathbb{P}(Z)$	$\Rightarrow$	$X \in \mathbb{P}(Z)$
$x \leqslant y \land y \leqslant z$	$\Rightarrow$	$x \leqslant z$
$(x \leqslant y \land y \leqslant z),\ x = z$	$\Rightarrow$	$x = y \land y = z$
$(X \in \mathbb{P}(Y)),\ a \in X$	$\Rightarrow$	$a \in Y$
$(a = b),\ R(b, x)$	$\Rightarrow$	$R(a, x)$
($X \times Y$ が記述中に在る), $X \in \mathbb{P}(Z)$	$\Rightarrow$	$X \times Y \in \mathbb{P}(Z \times Y)$
$\max(X) \leqslant \max(Y) \land \min(Y) \leqslant \min(X)$	$\Leftarrow$	$X \in \mathbb{P}(Y)$
$(S \in \mathbb{P}(Y),\ y \in Y),\ S = \emptyset$	$\Rightarrow$	$\neg(y \in S)$
$r[\{x\}] = \emptyset \land r[\{x\}] \in \mathbb{P}(\mathrm{dom}(r^{-1}))$	$\Leftrightarrow$	$\neg(x \in \mathrm{dom}(r))$
$(r \in \mathbb{P}(a \times b),\ (r;\ r^{-1}) \in \mathbb{P}(b)),\ y \in r[\{x\}]$	$\Leftrightarrow$	$y = r(x)$
$(r \in \mathbb{P}(a \times b),\ (r;\ r^{-1}) \in \mathbb{P}(b)),\ y \in r[\{x\}]$	$\Leftrightarrow$	$\{y\} = r[\{x\}]$
(id(X) または id(Y) が記述中に在る), $X \in \mathbb{P}(Y)$	$\Rightarrow$	$\mathrm{id}(X) \in \mathbb{P}(\mathrm{id}(Y))$
$x \in \{a_1, a_2, \ldots\}$	$\Leftrightarrow$	$x = a_1 \lor x = a_2 \lor \ldots$

著者略歴

中村 丈洋 (なかむら たけひろ)

生誕: 1983年 4月28日
出身: 新潟県 村上市 山辺里

2002年 3月 新潟県立 村上高等学校 卒業
2002年 4月 電気通信大学 電気通信学部 情報通信工学科 入学
2006年 3月 同上 卒業
2006年 4月 電気通信大学大学院 電気通信学研究科 情報通信工学専攻
博士前期課程 入学
2008年 3月 同上 修了
2008年 4月 電気通信大学大学院 電気通信学研究科 情報通信工学専攻
博士後期課程 進学
2013年 3月 同上 単位取得のうえ退学
2013年 4月 株式会社シフト 入社 現在に至る

形式手法，再利用，自動化，ソフトウェア検証の研究開発に従事.
情報処理学会会員