特別研究報告

題目

JML による仕様記述に対する Alloy Analyzer を用いた検証手法の提案 ―在庫管理プログラムに対する適用―

指導教員
楠本 真二 教授

報告者
森 恵弥佳

平成 25 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 24 年度 特別研究報告

JML による仕様記述に対する Alloy Analyzer を用いた検証手法の提案
—在庫管理プログラムに対する適用—

森 恵弥佳

内容梗概

ソフトウェア開発の重要概念である契約による設計（Design by Contract，以下 DbC とする）では，各クラス・メソッドが満たすべき仕様を表明として明確に定義することでソフトウェアの仕様理解の補助や，プログラム検証を可能にする．また表明の記述が十分であれば，メソッドの使用者が実装の詳細を考慮する必要がなくなり，情報隠蔽の度合いやモジュール性を高めることができる．DbC を実現するための具体的な言語として，Java のソースコード中に表明を記述するための JML（Java Modeling Language）などが存在する．また，オブジェクト指向モデリングに影響を受けた形式仕様記述言語として Alloy が存在する．Alloy は自動解析ツール Alloy Analyzer を備えており，記述したモデルに対して表明の検証を行うことができる．

本研究では，JML で記述された表明に関する欠陥を検出あるいは発見しやすくする手法を提案する．この手法は二つの要素からなっている．一つ目は，洗練の程度を可視化することで洗練の不十分さを使用者に発見しやすくするものである．まず，手動で JML を Alloy に変換する．その Alloy 記述を使用して，メソッドの表明が全て満たされたときのオブジェクトの例を Alloy Analyzer を用いてグラフとして出力する．二つ目は，実装と表明との不一致を検出するものである．まず，手動で JML と Java で書かれたソースコードを Alloy に変換する．その Alloy 記述を使用して，メソッドの実装が表明を満たさないときのオブジェクトの例を Alloy Analyzer を用いてグラフとして出力する．

適用実験はソフトウェア開発のための古典的な共通問題である在庫管理プログラムに対して行った．このプログラムは過去に研究グループが作成したものであり，Java で記述されている．また JML による表明も付与されている．実験の結果，8ヶ所の表明に関する欠陥を検出することができ，手法の有用性を確認できた．また，今回の手法では対応できなかった JML および Java の構文，Alloy Analyzer の解析の制限や記述および検証にかかるコストについても考察した．

主な用語

Alloy，JML，契約による設計，設計検証，仕様の洗練

目 次

1　まえがき

オブジェクト指向のソフトウェアの開発手法において，契約による設計（Design by Contract，以下 DbC とする）[1] と呼ばれる手法がある．DbC は，各クラス・メソッドが満たすべき仕様を表明として明確に定義しプログラム中に記述することで，ソフトウェアの仕様理解の補助やプログラム検証を可能にするものである．

記述された表明が不十分なものであった場合，実装者に仕様が正しく伝わらないことが考えられる．さらに「欠陥の責任がある箇所を特定可能」,「クラス・メソッドの利用者は実装を読む必要が無い」といった DbC の利点を十分に発揮させることが困難となる．また，実装が表明の記述を満たしていない場合は，実装の誤りと考えられる．あるいは，実装の段階で修正がなされたが表明には反映されていないという場合もあり，この場合も「クラス・メソッドの利用者は実装を読む必要が無い」という利点を十分に発揮させることが困難となる．よって表明の記述は必要十分であることが必要であり，不十分な記述は洗練 [2] し，実装と表明との不一致は修正する必要がある．しかし，表明が不十分であることを判定するのは困難である．

DbC を実現するための具体的な言語として，Java のソースコード中に表明を記述するための JML（Java Modeling Language）[3] などが存在する．JML で記述された表明が成り立つかを検査するツールの一つに ESC/Java2[4] があるしかし，これは不必要な警告も多く，また完全ではないことが知られている [5].

そこで本研究では，JML で記述された表明の洗練の程度を可視化することで洗練の不十分さを発見しやすくし，また JML で記述された表明と実装との不一致を検出する手法を提案する．そのために Alloy と Alloy Analyzer を使用した．Alloy[6] は集合と関係からなる一階述語論理に基づいたオブジェクト指向モデリングのための形式仕様記述言語である [7, 8]. Alloy Analyzer は，Alloy の自動解析ツールであり，記述したモデルに対して表明の検証を行うことができる．JML 記述の中で本手法の対象としたのは，不変条件，事前条件と事後条件であり，これは DbC の中心となるものである．手順として，まず JML と実装を Alloy に変換する．さらにその記述を使用して表明が実装を満たす場合の論理式および表明がすべて満たされる場合の論理式を Alloy で記述する．そして Alloy Analyzer で解析し，充足例・反例を出力する．

この手法の有効性を検証するために，ソフトウェア開発のための古典的な共通問題である在庫管理プログラムを用いて適用実験を行った．このプログラムは研究グループが過去に Java で作成したもの [9] であり，また JML による表明も付与されている．実験結果の一部は，文献 [10] でも取り上げている．

結果として，2ヶ所の表明の洗練が不十分さ，および 6ヶ所の実装と表明との不一致を指摘

することができた．また，今回の手法では対応できなかった JML および Java の構文，Alloy Analyzer の解析の制限や記述および検証にかかるコストについても考察する．

以降，2 章では研究の背景となる諸技術と関連研究について述べる．3 章では提案手法について述べ，4 章では適用実験とその結果についての考察を行う．最後に 5 章で本研究のまとめを述べる．

2　背景

本章では研究の背景となる諸技術と関連研究について簡単に述べる．

2.1　契約による設計 (DbC)

DbC とは，オブジェクト指向ソフトウェアにおける重要な開発手法である [1]．DbC は，現実社会における契約を模して，クラスとそのクラスの利用者の間の責務を明確に定義することで，信頼性の高いシステムを設計するというものである．また，クラス単位の再利用性を向上させることにもなる．契約違反が発生した場合，クラスとクラスの利用者のうちのどちらに責任があるのかが明白となる．このような責任の分離は開発者ごとの作業の分担を明確にし, ソフトウェアの欠陥の原因を切り分けるのに役立つ．あるいは，クラスが自分自身に課せられた契約をすべて満たす状態を，そのクラスが正しいと定義できる．またどこまでがクラスの責務であり，どこからはクラスの責務でないかを明確化することにもなるため，クラスにとって不必要な処理かどうかも判断できる．

契約は，ソースコード中に表明として記述される．また，表明はメソッドのふるまいを表す文書としても機能し，クラスの利用者は実装の詳細を考慮せずにすむ．ツールを使用して表明とクラスやメソッドの情報を抽出すれば，簡易な仕様書を作成できる．

主な表明には以下のものがある．

不変条件 クラスに付与される．クラスのインスタンスが生存期間中に満たさなければならない条件．クラスとクラスの利用者の両方の責務である．フィールドの値の制限などが該当する．

事前条件 メソッドに付与される．メソッドの実行前に満たされていなければならない条件．メソッドの呼び出し側の責務である．引数に対する条件などが該当する．

事後条件 メソッドに付与される．メソッドの実行後に満たされていなければならない条件．メソッド側の責務である．戻り値や実行後のフィールドの値に対する条件などが該当する．

2.2　JML

Eiffel[1] などの言語は DbC を言語レベルで取り入れているが，Java はそうではない．JML[3] は，Java で DbC を実現するための言語である．JML はソースコード中にコメントとして表明を記述する．文法は Java を踏襲したものであり，Java が記述できるのであれば初心者でも習得しやすいという特徴を持つ．

```
 1 Class Account{
 2     int balance;
 3     /*@ invariant balance >= 0; @*/
 4
 5     /*@ requires val > 0; @*/
 6     /*@ ensures balance == \old(balance) + val @*/
 7     public void deposit(val : int){
 8         balance = balance + val
 9     }
10 }
```

図 1: JML による表明記述の例

図 1 に JML による表明記述の例を示す．3 行目の “invariant” は不変条件を表し，クラス Account のフィールド変数である balance は非負であることを表している．5 行目の “requires” は事前条件を表し，メソッド deposit を呼び出す場合は引数 val を正としなければならないことを表している．6 行目の “ensures” は事後条件を表し，実行後の balance の値は，実行前の balance の値に引数 val の値を足したものであることを表している．ここで “\old(変数名)” は事後条件中で使用され，メソッド実行前のその変数の値を表すものである．これによりメソッド実行前と実行後の変数の値を区別し，実行前の状態と実行後の状態の関係を記述することができる．

この他に JML 独自の構文として，事後条件中でメソッドの戻り値を表す “\result”，Type 型の変数 t のうち boolean 式 1(t) を真にするものはすべて boolean 式 2(t) を真にするということを表す “(\forall t Type; boolean 式 1(t); boolean 式 2(t))”，そして Type 型の変数 t のうち boolean 式 1(t) を真にするものの中に boolean 式 2(t) を真にするものが存在するということを表す “(\exists t Type; boolean 式 1(t); boolean 式 2(t))” などがある．

2.3 Alloy

Alloy は，集合と関係からなる一階述語論理に基づいたオブジェクト指向モデリングのための形式仕様記述言語であり，Z 言語をもとにしている [7].

Alloy は，自動で検証が行えること (軽量形式手法 [6]) を主眼において設計されており，自動解析ツール Alloy Analyzer[11] を備えている．Alloy Analyzer により，Alloy で記述したモデルや追加した条件を満たす例 (充足例) や，記述したモデルを満たすもののうち，追加した条件を満たさない例 (反例) を解析し出力することができる．このとき実行コードやテ

```
1 module test/account
2
3 sig Account {
4       balance : one Int
5 }
6 pred invariant[a:Account] {
7       a.balance >= 0
8 }
9
10 pred Account.deposit[a':Account,val:Int] {
11       a'.balance = this.balance.plus[val]
12 }
13 pred deposit_pre[val:Int] {
14       val > 0
15 }
16
17 pred show_deposit[a,a':Account,val:Int] {
18       invariant[a] && deposit_pre[val] && a.deposit[a',val]
19 }
20 run show_deposit
21
22 assert check_deposit {
23       all a,a':Account, val:Int |
24            invariant[a] && a.deposit[a',val]
25                 => invariant[a']
26 }
27 check check_deposit for 5
```

図 2: Alloy の記述例

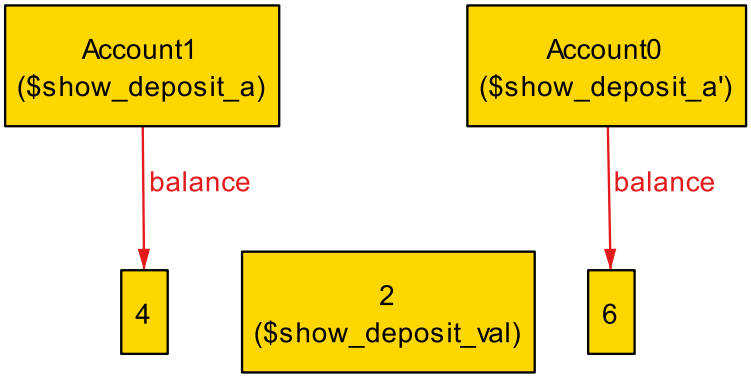


図 3: Alloy Analyzer の解析結果の例 (show_deposit)

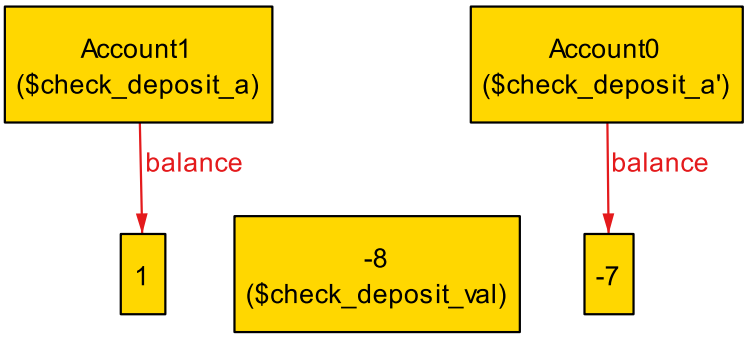


図 4: Alloy Analyzer の解析結果の例 (check_deposit)

ストケースなどは必要ない．出力はグラフなどの形式でなされるため，モデルの設計者は記述したモデルの視覚的フィードバックを得ることがいえる．

解析の際にはスコープという非負整数で解析範囲を制限することで，範囲内を網羅的に解析できる．スコープは，モデルの中で定義された各オブジェクトの個数の上限を表している．バックエンドには外部の SAT ソルバ [12] を利用しており，用意された複数の SAT ソルバから選択できる．Alloy Analyzer は，Alloy で記述されたモデルを充足可能性問題 (SAT) に変換して SAT ソルバに渡し，結果を受け取って視覚的に出力する役目を果たしている．

小スコープ仮説 [7] により異常な充足例は小さいスコープに存在すると考えるため，比較的小さいスコープでも反例や特徴的な充足例を検出することができる．また，スコープ単調性 [7] により，あるスコープで反例が見つからなかった場合，それより小さいスコープで反例が存在しないことがわかる．

図 2 に Alloy で記述したモデルの例を示す．これは図 1 をモデルにしたものである．3 行目の “sig” は，シグネチャの宣言を表す．シグネチャとはオブジェクトであり，図 1 のクラスに対応する．4 行目はフィールドであり，図 1 と同じく Account は balance という名前の整数を所持している．6 行目の “pred” は述語の宣言を表す．述語は真か偽を返す関数である．述語 invariant は図 1 の不変条件に対応している．10 行目の述語 Account.deposit は図 1 のメソッド deposit に，13 行目の述語 deposit_pre はメソッド deposit の事前条件に対応している．

17 行目の述語 show_deposit は，メソッド deposit が適切な状態で実行されたときの状態を表現している．引数 a は実行前の Account，a’ は実行後の Account を表している．show_deposit 全体では，実行前に不変条件と事前条件が成り立っている状態で，deposit が実行されたことを表す．20 行目の “run” はコマンドの一つであり，記述したモデルのうち指定した述語を真にする例 (充足例) の解析を Alloy Analyzer に指示する．ここでは show_deposit の充足例を求めており，出力例は図 3 で，他にもいくつかの充足例を順に表示する．

22 行目の “assert” はアサーションを表す．アサーションは成り立つかどうかを検査したい述語である．アサーション check_deposit は，deposit が不変条件を保存することを表現している．具体的には「考えられるすべての状態に対して，実行前に不変条件が成り立っている状態で deposit が実行されたなら実行後も不変条件が成り立つ」ということを表す．27 行目の “check” はもう一つのコマンドであり，記述したモデルのうち指定したアサーションを偽にする例 (反例) の解析を Alloy Analyzer に指示する．ここでは check_deposit の反例を求めている．“for 5” というのはスコープの指定であり，ここでは同時に存在する Account の個数の上限を 5 に制限して解析を行うことを意味する．スコープの指定は run コマンドでも同様に行える．指定しなかった場合のデフォルト値は 3 である．この解析の出力例は図 4 で，他にもいくつかの反例を順に表示する．図 4 より，check_deposit では事前条件の成立

を前提に含めなかったために反例が存在することが分かる．

またここでは使用しなかったが，シグネチャを返す関数の宣言を表す “fun” などがある．

2.4 関連研究

2.4.1 ESC/Java2

ESC/Java2[4] は，Java プログラムに対する静的検証器である．ESC/Java2 は Java ソースコードを入力とし，NullPointerException などの例外が発生する可能性がある箇所に対して警告を出力する．また，JML などで記述された表明と Java ソースコードの整合性の検証も行うことができる．ESC/Java2 は対象プログラムの検証を，対象プログラムを述語論理に変換し，その充足不能性を判定することによって行う．

JML の構文にはいくつかのレベルが設定されている [13] が，ESC/Java2 はそのすべてに対応しているわけではない [5]．また，Java1.5 以降の文法に対応していない．

ESC/Java2 は不必要な警告も多く，またすべての契約違反を検出できるわけではないことが知られている．

2.4.2 その他

Alloy に関する研究については，JUnit 用のテストケースを生成するツールの開発 [14] や，システム再構成の際の信頼性を解析する実用的な手法に関する研究 [15] が行われている．また，Alloy の解析の高速化に関する研究 [16, 17] や Alloy を拡張し動的な性質をより記述しやすくしようとする研究 [18, 19] も存在する．さらに，OCL 付きの UML から Alloy を生成するツール [20] や，その逆変換を行うツール [21] も開発されている．

仕様の洗練については，洗練が必要となる不十分さや曖昧さを含むパターンを分類し，分類したものについてそれを解消するための枠組みを提案する研究 [22] が存在する．

Java 以外の言語の検査ツールには，CForge[23] や CBMC[24] があり，CForge は仕様記述のための言語として JML を拡張したものを使用している．また，C#を拡張し契約を記述・検証可能にした言語として Spec#[25] がある．

3　提案手法

本章では，まず使用する用語の定義を行い，次に提案手法について詳細に述べる．

3.1　定義

本稿で提案する手法では，次の二つを表明に関する欠陥であるとする．

- 表明の洗練の不十分さ
- 実装と表明との不一致

それぞれについて，次小節以降で詳しく説明する．

3.1.1　表明の洗練の不十分さ

表明の洗練の不十分さとは，表明がメソッドの主要なふるまいを十分に表現できていない状態を指す．

これには以下のものが該当する．

- 不変条件が不十分であり，クラスの性質を十分に表現できていない
- 事前条件が不十分であり，メソッドのふるまいを十分に表現できていない
- 事後条件が不十分であり，メソッドのふるまいを十分に表現できていない

自然言語で記述された要求から仕様を策定する場合，一度で必要十分な表明を記述することは困難である．表明の洗練が不十分であると，実装者が参照した際に仕様が正しく伝わらず，後の工程での不具合の発生を招く恐れがある．また 2.1 節で述べたような DbC における利点を十分に発揮することができなくなる．

よって，表明の洗練の不十分さは検出し解消しなければならないが，機械的に判断することは困難である．また，表明に使用される論理式は複雑なものもあり，さらにはメソッドの引数となるオブジェクトの不変条件など，関連する表明が複数のファイルに分散していることもある．したがって，人の目で直接確認することも難しい．

表明の洗練の不十分さとして，他に以下のものも考えられる．しかし，以下のものはメソッドが対応すべき状態が網羅的にわかっていることが必要であるため，対応が困難である．よって，本稿の手法では対象外とする．

- 不変条件が過剰であり，クラスが表現すべき状態を除外してしまう
- 事前条件が過剰であり，メソッドが対応すべき状態を除外してしまう

3.1.2 実装と表明との不一致

実装と表明との不一致とは，実行前の不変条件・事前条件が成立していても実装が実行後の不変条件・事後条件を成立させることができないことを指す．

これには以下のものが該当する．

- 不変条件・事前条件が不十分であり，実装が事後条件・不変条件を満たさない
- 実装が誤っており，事後条件・不変条件を満たさない
- 事後条件が誤っており，メソッドのふるまいと一致しない

単に実装を行う際に誤ってしまったという場合もあるが，実装の時点で欠陥が検出・修正されたが，表明には修正が反映されていない場合などもある．同様に，実装後に表明が変更されたが実装には反映されていない場合もある．実装が正しく表明が誤っている場合は表明の検証が意味をなさない．表明が正しく実装が誤っている場合はソフトウェアの欠陥であるといえる．いずれにしろ修正する必要があるが，その際にはどちらが誤っているのか，あるいは両方とも誤っているのかを判断するために，より上位の仕様が必要となる．

上記で説明した欠陥は，Hoare triple[26] が示すソフトウェアの正しさが成り立たない場合の一部である．

このソフトウェアの正しさが成り立たない場合として，他に以下のものも考えられる．しかし以下のものはメソッドの実行前と実行後以外の状態に言及するものであるため，該当する箇所が膨大となり，検査が困難となる．よって本稿の手法では対象外とする．

- 事前条件が満たされない状態でメソッドが呼び出される

3.2 検証方法

前節で説明した 2 種類の表明に関する欠陥に対して，Alloy Analyzer を利用した次の手法を提案する．

- 表明の洗練の不十分さを発見しやすくするために，表明の洗練の程度を可視化する
- 実装と表明との不一致を検出する

手順の流れは図 5 の通りである．

実装と表明との不一致の検出に関しては，Alloy Analyzer の出力により不一致の有無を瞬時に判断することができる．しかし，表明の洗練の程度の可視化に関しては，Alloy Analyzer が出力したグラフを人の目で確認する必要があるため，判断には時間を要する．また，人の目で判断するため表明の洗練の程度の可視化によって実装と表明との不一致が発見されるこ

ともある．本報告では説明の流れの都合上表明の洗練の程度の可視化から説明するが，先述の確実性と所要時間上の理由により実際に用いる場合は実装と表明との不一致の検出先に行った．また，表明の洗練の程度の可視化と実装と表明との不一致の検出の両方で指摘された欠陥は，実装と表明との不一致によって検出されたものとして扱う．

それぞれについて，次小節以降で詳しく説明する．

3.2.1 洗練の程度を可視化

洗練の程度を可視化する手法について詳しく説明する．まず，入力として使用するのは JML による表明である．出力となるのは，表明がすべて成立したときの状態を表現したグラフである．ここでいう状態とは，引数と実行前のオブジェクトと戻り値と実行後のオブジェクトの値である．表明の洗練が不十分な場合は出力されるグラフに意図していない状態が含まれるため，人の目で表明の洗練が不十分かどうかを判断することが容易となる．この手法により発見できる表明の欠陥は，3.1.1 節で対象としたものである．

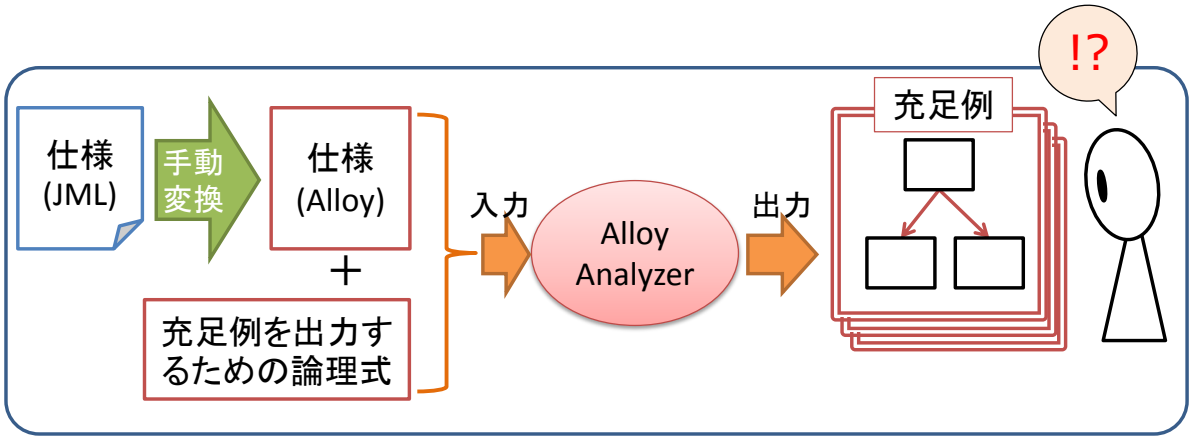
流れは以下の通りである．これをまとめたものが図 5(a) である．

1. JML を Alloy に手動で変換する．
2. 1. で変換した記述を使用して，表明同士の関係を Alloy の述語として記述する．
3. run コマンドで Alloy Analyzer に解析を指示し，結果を表示させる．

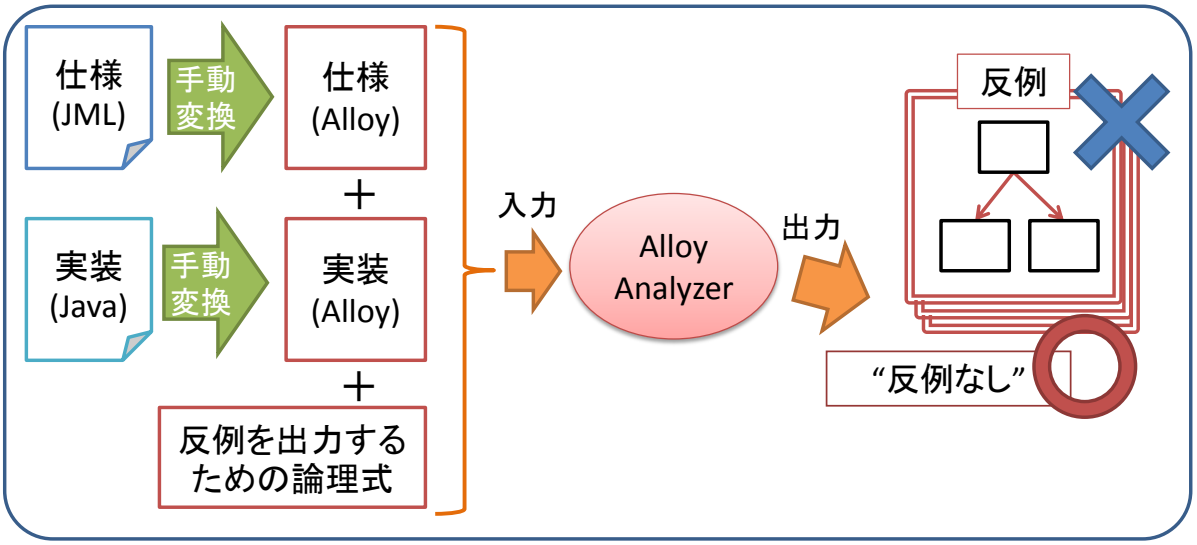
手順 1. の JML から Alloy への変換方法については 3.3.1 節で詳しく説明する．

手順 2. の「表明同士の関係」の記述は図 6 のものである．ここで “A” はクラス名，“a” は実行前のクラス A のオブジェクト，“a’ ” は実行後のクラス A のオブジェクト，“methodA” はクラス A のあるメソッドとする．さらに “methodA” の引数名は “para”，引数の型名は “Para”，戻り値を “return”，戻り値の型名を “Return” とする．“Para/inv[para]” は引数の不変条件の成立を，“inv[a]” はメソッドの実行前のオブジェクトの不変条件の成立を，“methodA_pre[para]” はメソッドの事前条件の成立を，“methodA_post[a,a’,para,return]” はメソッドの事後条件の成立を，“inv[a’]” はメソッドの実行後のオブジェクトの不変条件の成立を，“inv[return]” は戻り値の不変条件の成立を表している．引数の個数などはメソッドに合わせて変更する．

述語 “check_methodA” は上記の条件をすべて論理積 “&&” で掛け合わせたものである．したがって,「“methodA” に関係する表明がすべて成立する」ことを表している．これを満たす状態を，5 行目の run コマンドにより Alloy Analyzer を使って解析する．



(a) 洗練の程度を可視化



(b) 実装と表明との不一致の検出

図 5: 提案手法の流れ

```
1 pred check_methodA[a,a':A, para:Para, return:Return] {
2     Para/inv[para] && inv[a] && methodA_pre[para]
3         && methodA_post[a,a',para,return] && inv[a'] && inv[return]
4 }
5 run check_methodA
```

図 6: 表明同士の関係の Alloy 記述

3.2.2 実装と表明との不一致の検出

実装と表明との不一致の検出する手法について詳しく説明する．まず，入力として使用するのは JML による表明とメソッドの実装である．出力となるのは実装が仕様を満たすかどうかと，満たさない場合はそのときの状態を表現したグラフである．ここでいう状態とは前小節のものと同様である．実装が仕様を満たさない状態がグラフとして出力されるため，人の目でどこに問題があるのかを判断することが容易となる．この手法により発見できる表明の欠陥は，3.1.2 節で対象としたものである．

流れは以下の通りである．これをまとめたものが図 5(b) である．

1. JML とメソッドの実装を Alloy に手動で変換する．
2. 1. で変換した記述を使用して，実装と表明の関係を Alloy のアサーションとして記述する．
3. check コマンドで Alloy Analyzer に解析を指示し，結果を表示させる．

手順 1. の JML から Alloy への変換方法については 3.3.1 節で，Java から Alloy への変換方法については 3.3.2 節で詳しく説明する．

手順 2. の「実装と表明の関係」の記述は図 7 のものである．ここで図 6 と同様に，“A” はクラス名，“a” は実行前のクラス A のオブジェクト，“a’” は実行後のクラス A のオブジェクト，“methodA” はクラス A のあるメソッドとする．さらに “methodA” の引数名は “para”，引数の型名は “Para”，戻り値を “return”，戻り値の型名を “Return” とする．“Para/inv[para]” は引数の不変条件の成立を，“inv[a]” はメソッドの実行前のオブジェクトの不変条件の成立を，“methodA_pre[para]” はメソッドの事前条件の成立を，“a.methodA[a’,para,return]” は a に対するメソッドの実行を，“methodA_post[a,a’,para,return]” はメソッドの事後条件の成立を，“inv[a’]” はメソッドの実行後のオブジェクトの不変条件の成立を，“inv[return]” は戻り値の不変条件の成立を表している．引数の個数などはメソッドに合わせて変更する．

アサーション “methodA” は「実行前に不変条件と事前条件が成立している状態でメソッドを実行すれば，実行後に不変条件と事後条件が成立する」ことを表す．これは,「実装が表明を満たす」ことを表している．これが成立しない状態を，6 行目の check コマンドにより Alloy Analyzer を使って解析する．

ただし，図 7 のアサーションの含意の左辺が偽である場合も，式全体が真となってしまい反例は出力されない．このとき，実装が仕様を満たしている場合との識別ができない．

```
1 assert methodA {
2     all a,a':A, para:Para, return:Return |
3         Para/inv[para] && inv[a] && methodA_pre[para] && a.methodA[a',para,return]
4             => methodA_post[a,a',para] && inv[a']
5 }
6 check methodA
```

図 7: 実装と表明の関係の Alloy 記述

```
1 pred show_methodA[a,a':A, para:Para, return:Return] {
2     Para/inv[para] && inv[a] && methodA_pre[para] && a.methodA[a',para,return]
3 }
4 run show_methodA
```

図 8: 確認に必要な Alloy 記述

含意の左辺が偽になる原因は，以下のものが考えられる．

- 不変条件，事前条件，またはメソッドの実装の Alloy 記述に誤りがある
- 不変条件・事前条件が過剰であり，充足例が存在しない

上記の状態は明らかに誤りであり，実装が仕様を満たしている場合との識別する必要がある．そのために図 8 の記述も実行し充足例が存在することを確認する必要がある．

図 8 の 2 行目は，図 7 の 3 行目と同一であり，アサーション “methodA” の含意の左辺である．これを満たすものが存在するかどうかを，4 行目の run コマンドにより Alloy Analyzer を使って解析する．

3.3 変換方法

ここでは，前小節で省略した JML と Java の Alloy への変換方法について詳細に述べる．

3.3.1 JML から Alloy への変換

JML から Alloy への変換に関しては，どちらも論理式で構成されていることから比較的容易である．対応する記述例は表 1 の通りである．表 1 の最後の列は，\old の変換例である．JML の \old は対応するものが Alloy に存在せず，個別に対応する必要がある．あるいはデータ構造に取り入れる方法もあるが，今回は採用しなかった．

表 1: JML と Alloy 間の対応する記述例

JML の記述	Alloy の記述
a.contains(b)	b in a
(\exists t Type; boolean 式 1(t); boolean 式 2(t))	some t:Type | boolean 式 1(t) && boolean 式 2(t)
(\forall t Type; boolean 式 1(t); boolean 式 2(t))	all t:Type | boolean 式 1(t) => boolean 式 2(t)
(\sum t Type; a.contains(t); boolean 式 (t))	all t:a | boolean 式 (t)
!\old(allitemlist.isEmpty()) ==> !allitemlist.equals(c.getContainedItem());	s.allitemlist != none => s'.allitemlist != c.itemlist

変換できなかった JML の構文は，\typeof() と \type() であり，これはそれぞれ引数の集合の要素の型と引数の型を返すものである．また，文字列型の値の比較を行う “equals()” と参照の比較を行う “==” を書き分けと，集合型の null と空集合の書き分けができなかった．本手法では，文字列の比較は値の比較を表し，集合型が空である場合は空集合を表しているとする．

3.3.2 Java から Alloy への変換

Java から Alloy への変換に関しては，ソースコードから論理式への変換であることから比較的困難である．ソースコードのデータ依存関係や制御依存関係をもとに，オブジェクトの実行前と実行後に成り立つ条件を Alloy の述語として記述する．Alloy の述語の引数にはメソッドの引数だけでなく，メソッドの実行後のオブジェクトと戻り値を加える．対応する記述例は表 2 の通りである．

変換の際の注意点を以下に挙げる．

1. Alloy では限量子に束縛されない変数を使用できないため，メソッドの実行前の状態と実行後の状態の間に中間状態が必要な場合は工夫が必要

2. Alloy では，再帰やループに対応する構文が無いため，完全には変換できないものもある

2. について具体的に説明する．ループのうち Alloy に変換できないのはループの実行順序によって事後状態が変化するループであり，実行順序が事後状態に影響を与えないループは Alloy に変換できる．実行順序が事後状態に影響を与えないループとは，検索のためのループやすべての要素を一律に更新するためのループである．

表 2: Java と Alloy 間の対応する記述例

Java の記述	Alloy の記述
コンストラクタ	

public Item(String n, int a) {
  name = new String(n);
  totalamount = a;
}
	pred Item.constructor[n:Str,a:Int] {
	  this.name = n
	  this.totalamount = a
	}

setter	

public void setAmount(int num) {
  totalamount = num;
}
	pred Item.setAmount[i':Item, a:Int] {
	  i'.name = this.name
	  i'.totalamount = a
	}

getter(while 文を使用)	

public int getAmount(String name) {
  Iterator i=itemlist.iterator();
  while(i.hasNext()) {
  Item item = (Item)i.next();
  if(item.getName().equals(name)) {
  return item.getAmount();
  }
  }
  return 0;
}
	fun ContainerItem.getAmount[n:Str]:Int {
	  n in this.itemlist.name
	  => (this.itemlist <: name).n.totalamount
	  else 0
	  }

if-else 文と関数呼び出し	

if(item.getAmount() > num) {
  item.setAmount(item.getAmount()-num);
} else {
  item.setAmount(0);
}
	i.totalamount > a
	  => i.setAmount[i',i.totalamount.minus[a]]
	  else i.setAmount[i',0]

変換できるループ 1：検索

まず，検索のためのループの例について説明する．表 2 の getter は while 文を使用している特殊な getter であるが，この while 文はフィールド itemlist から，getName() の戻り値が引数 name と等しいものを検索するためのループである．このメソッドを持つクラスには「itemlist 内で品名 (getName() が返す値) は一意である」という不変条件が存在する．よって，イテレータがどのような順序で itemlist の要素を取り出したとしても，return 文によって返される値は変わらない．したがって，表 2 のように Alloy でも記述することができる．

ここで “<:” は制限演算子であり，右項の関係の定義域を左項に制限するものである．そして “(this.itemlist <: name).n” で,「このメソッドを適用したオブジェクトの itemlist の要素のうち name フィールドが n であるもの」を表す．

制限演算子により Java よりも簡潔に検索のためのループを記述できる．

変換できるループ 2：一律な処理

次に，すべての要素を一律に更新するためのループについて説明する．図 9 は引数 c(ContainerItem のインスタンス) の itemlist と実行前のフィールド itemlist を合計したものを実行後の itemlist とするためのコード片であり，外側の while 文が本体となる．内側の while 文は itemi と品名が等しい要素をフィールド allitemlist から検索し，その要素を更新するためのループであり，これは図 10 の 1–4 行目が対応する．図 9 の 12–15 行目は，引数 c の itemlist には含まれているが実行前の allitemlist には含まれていなかった要素を実行後の allitemlist に加えるためのものであり，これは図 10 の 13–16 行目が対応する．

Java では，実行前と実行後のオブジェクトは同一のものが変化したものであり，変化に関する記述だけが必要である．しかし，Alloy では実行前のオブジェクトと実行後のオブジェクトは全く別物となるため，変化しないもの，つまり実行前に含まれており変化せず実行後も含まれている要素や実行前は含まれず実行後も含まれない要素に対する記述も必要である．図 10 の 5–12,17–21 行目がこの記述に該当する．

図 10 の 5,6 行目は，実行前の allitemlist に含まれていなかった要素が実行後に含まれないようにするための記述である．11,12,17–21 行目も同様である．また図 10 の 7–10 行目は，実行前の allitemlist には含まれているが引数 c の itemlist には含まれていなかった要素を実行後の allitemlist に加えるためのものである．

以上のように更新のためのループでは，Alloy では実行前のオブジェクトと実行後のオブジェクトは全く別物となるために記述の量が増加し，また複雑となる．

変換できないループ

変換できないループについて説明する．図 11 のメソッドは引数の注文の数量だけ倉庫から取り出す操作を行う．外側の while 文は，allitemlist 内に注文された商品が存在するかどうかを検索し，存在した場合は注文の数量だけ在庫を減らすためのループである．これは検索のためのループであるため Alloy で記述可能であり，図 12 の 2–6 行目が対応する．しかし図 11 の内側のループは containerlist 内の先頭の要素から順に注文を満たすまで取り出していくためのループであり，リストをたどる順序が変われば実行後のリストの状態も変化する．

Alloy では，リストの取り扱いは複雑であるため，containerlist を初めとして各リストは集合として扱った．そのため，このようなループを完全に変換することができない．そこで，containerlist に対する次の 3 つの制約の論理積で代用する．

- 実行前と実行後の注文された商品の合計の差は，注文された数量と等しい (図 12 の 27–29 行目)
- 実行前と実行後で数量が減少する要素と変化しない要素がある (図 12 の 12–19 行目)
- 前項目の減少するもの要素「減少するがゼロにはならない」に該当する要素は高々一つである (図 12 の 21,22 行目)

この制約が while 文の代用となることについて，図 13 の具体例で説明する．図 13(a) は実行前の各要素が所持している注文された商品の個数である．注文数が 5 の場合先頭から順番に取り出せば，実行後は図 13(b) のようになる．注文数が 7 の場合先頭から順番に取り出せば，実行後は図 13(c) のようになる．どちらも前半の要素は実行後の数量がゼロであり，後半の要素は数量が実行前と同じであり，そのどちらでもないものはその境界に高々1 つ存在する．つまり，上記の 3 つの制約の論理積により，実際の商品がある順序でならんでいた際の図 11 の内側の while 文の実行を表現したことになる．

```
1  Iterator i = c.getContainedItem().iterator();
2  while(i.hasNext()){
3      Item itemi = (Item) i.next();
4      Iterator j = allitemlist.iterator();
5      while(j.hasNext()){
6          Item itemj = (Item) j.next();
7          if((itemj.getName()).equals(itemi.getName())){
8              itemj.setAmount(itemj.getAmount() + itemi.getAmount());
9              break;
10         }
11     }
12     if(!j.hasNext()){
13         Item tmpitem = new Item(itemi.getName(),itemi.getAmount());
14         allitemlist.add(tmpitem);
15     }
16 }
17 return;
```

図 9: 変換できるループの例 (Java)

```
1  all i,j:Item |
2    i in c.getContainedItem[] && j in this.allitemlist && i.getName[]=j.getName[]
3      => {one j':Item |j' in s'.allitemlist && i.getName[]=j'.getName[]
4        && j'.getAmount[] = i.getAmount[].plus[j.getAmount[]]}
5          &&{no j':Item |j' in s'.allitemlist && i.getName[]=j'.getName[]
6            && j'.getAmount[] != i.getAmount[].plus[j.getAmount[]]}
7  all i:Item |
8    i in c.getContainedItem[] && {all j:Item |j.getName[]=i.getName[] => j not in this.allitemlist}
9      => {one j':Item |j' in s'.allitemlist && i.getName[]=j'.getName[]
10       && j'.getAmount[] = i.getAmount[]}
11         &&{no j':Item |j' in s'.allitemlist && i.getName[]=j'.getName[]
12           && j'.getAmount[] != i.getAmount[]}
13 all j:Item |
14   j in this.allitemlist && {all i:Item |i.getName[]=j.getName[] => i not in c.getContainedItem[]}
15     => {one j':Item |j' in s'.allitemlist && j.getName[]=j'.getName[]
16       && j'.getAmount[] = j.getAmount[]}
17         && {no j':Item |j' in s'.allitemlist && j.getName[]=j'.getName[]
18           && j'.getAmount[] != j.getAmount[]}
19 all j':Item |
20     {no i:Item |(i in c.getContainedItem[] ||i in this.allitemlist) && i.getName[]=j'.getName[]}
21         => j' not in s'.allitemlist
```

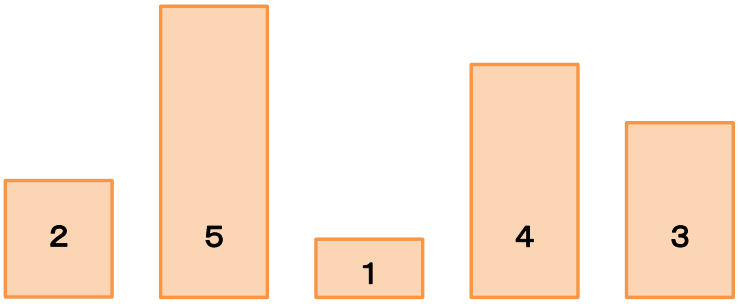
図 10: 変換できるループの例 (Alloy)

```
 1  public void deliveringOrder(Request r) {
 2    int n = r.getAmount();
 3
 4    Iterator i = allitemlist.iterator();
 5    while(i.hasNext()) {
 6      Item item = (Item)i.next();
 7      if(item.getName().equals(r.getName())) {
 8          item.setAmount(item.getAmount() - r.getAmount());
 9          Iterator j = containerlist.iterator();
10          while(j.hasNext()){
11              ContainerItem containeritem = (ContainerItem)j.next();
12              n = containeritem.shippingItem(r.getName(),r.getAmount());
13              if(n<=0) break;
14          }
15          break;
16      }
17    }
18    return ;
19  }
```

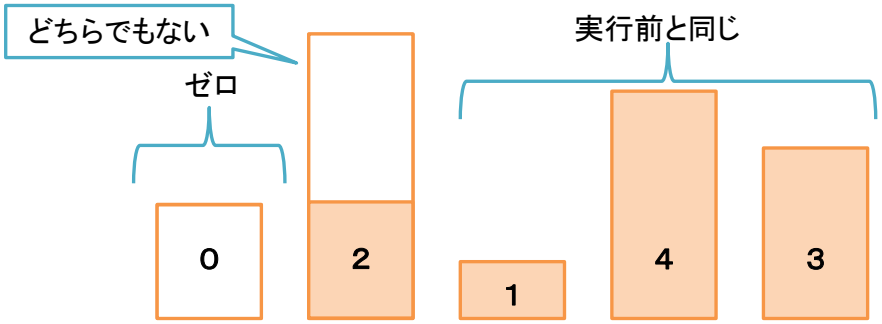
図 11: 変換できないループの例 (Java)

```
1  pred Storage.deliveringOrder[s':Storage, r:Request] {
2  one i,j:Item {
3    i in this.allitemlist && i.name = r.itemname
4    i.setAmount[j,i.totalamount.minus[r.amount]]
5    s'.allitemlist = this.allitemlist - i + j
6  }
7
8  all c:ContainerItem |
9    c in this.containerlist && c.getAmount[r.itemname] = 0
10     => c in s'.containerlist && { no c":ContainerItem |
11       c" in s'.containerlist && c != c" && c.containerID = c".containerID }
12 all c:ContainerItem |
13   c in this.containerlist && c.getAmount[r.itemname] != 0
14     => { { one c':ContainerItem, a,a':Int |
15       c.getAmount[r.itemname] >= a && a > 0
16         && c.shippingItem[c',r.itemname,a,a'] && c' in s'.containerlist }
17       && { one c':ContainerItem |c' in s'.containerlist && c.containerID = c'.containerID } }
18     ||{ c in s'.containerlist && { no c":ContainerItem |
19       c" in s'.containerlist && c != c" && c.containerID = c".containerID } }
20
21 lone c':ContainerItem |
22   c' not in this.containerlist && c' in s'.containerlist && c'.getAmount[r.itemname] != 0
23 all c':ContainerItem |
24   { no c:ContainerItem |c in this.containerlist && c.containerID = c'.containerID }
25     => c' not in s'.containerlist
26
27 ( sum c:this.containerlist |c.getAmount[r.itemname] )
28   .minus[( sum c':s'.containerlist |c'.getAmount[r.itemname] )]
29     = r.amount
30 }
```

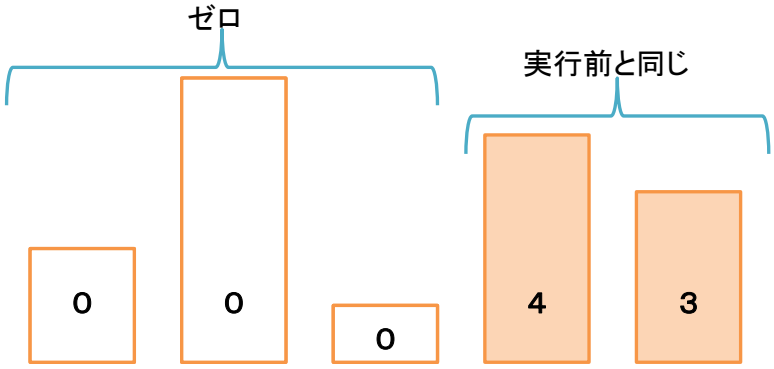
図 12: 変換できないループの例 (Alloy)



(a) 実行前の状態



(b) 注文数が 5 の場合



(c) 注文数が 7 の場合

図 13: ループの代用の具体例

4　適用実験

本章では，まず実験対象プログラムについての紹介し，次に実験結果の説明と考察を述べる．

4.1　対象プログラム

対象とするのは，ソフトウェア設計の共通問題である酒屋の在庫管理問題を実装したプログラムである．これは，著者の研究グループが過去に文献 [9] で設計・開発したものである．JML による不変条件，事前条件，事後条件および代入可否などの表明が付加されている．それらの表明は ESC/Java2 により検証されているが，2.4.1 節で述べたように完全な検証が行われたとはいえない．

規模は，クラスが 7 個，コンストラクタが 11 個，メソッドが 41 個である．各クラスの規模は表 3 のようになっている．JML の行数と Java の行数の和と，合計の行数が異なるのは，Java と JML が混合した行が存在するためである．

クラスの構成とふるまいについて，それぞれ次小節以降で詳しく述べる．

4.1.1　クラスの構成

対象プログラムは以下の 7 つのクラスからなる．図 14 にクラス図を示す．

- 注文の状態を表すための列挙型を表す StockState クラス (注文状態)
- 顧客情報を表す Customer クラス (顧客)
- 注文情報を表す Request クラス (注文)
- 商品情報を表す Item クラス (品目)
- 複数の品目を格納したコンテナを表す ContainerItem クラス (コンテナ)
- 複数のコンテナと，コンテナに格納された品目の合計を表す在庫リストを管理する Storage クラス (倉庫)
- 外部とのやり取り，変化への対応を担当する ReceiptionDesk クラス (受付係)

表 3: 対象プログラムの規模 (空行・コメント行を含む)

クラス名	Java の行数	JML の行数	総行数	コンストラクタ数	メソッド数
StockState	4	0	4	0	0
Customer	52	71	115	2	8
Request	78	107	175	3	9
Item	29	35	60	1	4
ContainerItem	75	72	141	2	6
Storage	112	134	240	2	7
ReceiptionDesk	75	67	136	1	7
合計	425	486	871	11	41

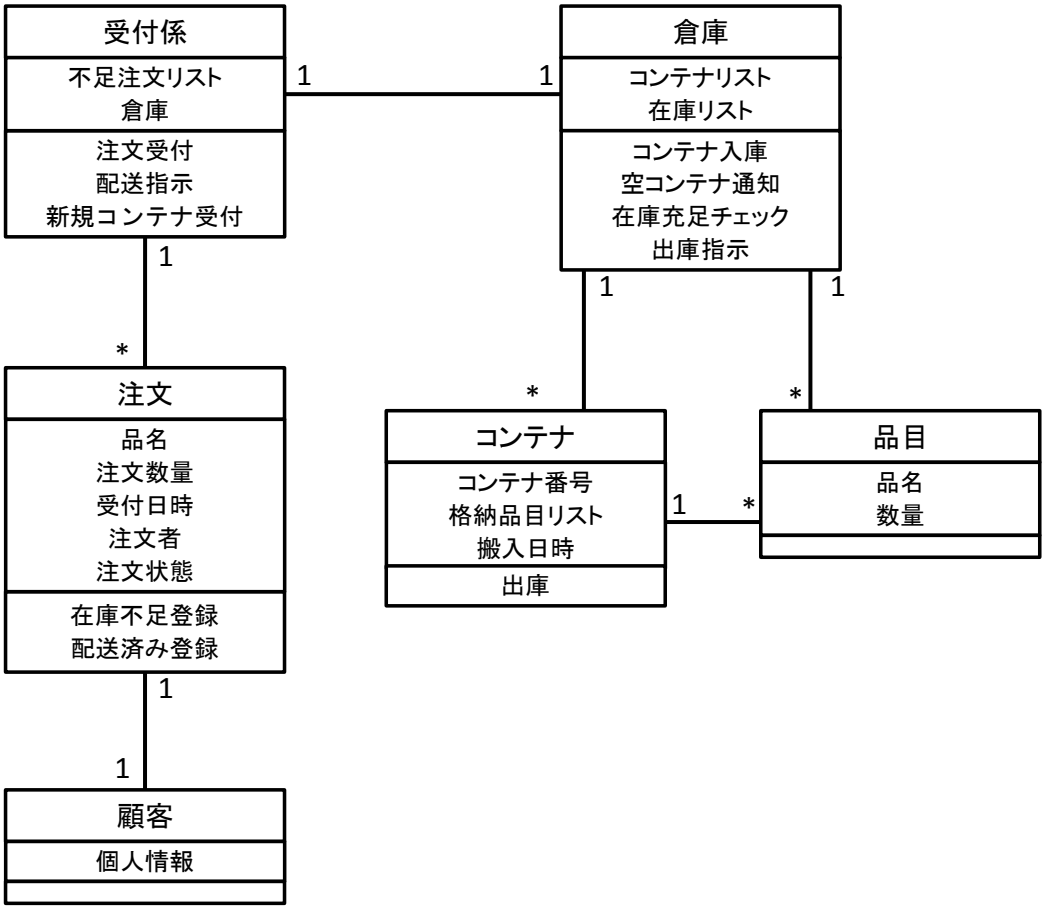


図 14: 対象プログラムのクラス図

4.1.2 ふるまい

対象プログラムのふるまいについて述べる．このプログラムには，注文の受付とコンテナの搬入の 2 つの機能がある．それぞれについて順に説明する．

注文の受付

注文に対応できるだけの在庫が存在する場合は以下のようになる．

1. 受付係が注文を受け付け，在庫で対応できるか確認する
2. 対応できる場合は，注文を満たすようにコンテナリストのコンテナから取り出す
3. コンテナの内容の変化にあわせて，在庫リストを更新する
4. 対応できたので，注文の情報は捨てる

注文に対応できるだけの在庫が存在しない場合は以下のようになる．

1. 受付係が注文を受け付け，在庫で対応できるか確認する
2. 対応できない場合は，未対応の注文として不足注文リストに追加する

コンテナの搬入

コンテナの搬入の際のふるまいは以下のようになる．

1. 受付係がコンテナを受け付け，倉庫のコンテナリストに追加する
2. 在庫状態の変化にあわせて，在庫リストを更新する
3. 在庫状態の変化により未対応の注文に対応できる可能性があるので，受付係が不足注文リストを順に確認する
4. 対応できる注文があれば，その分をコンテナから取り出し在庫リストを更新する

4.2 検査の対象とする要素

対象プログラムのうち，本実験で検査の対象とするのは 6 個のクラス，5 個のコンストラクタと 18 個のメソッドに付与された，6ヶ所の不変条件，21ヶ所の事前条件と 23ヶ所の事後条件である．

コンストラクタは，各クラスの主要なもの 1 つずつのみとした．また ReceiptionDesk クラスのコンストラクタには対象とする表明が記述されていなかったため対象外とした．

メソッドについては，単純にフィールドの値を返すだけの 19 個の setter は，自明であるため対象外とした．また，実装が “return true” のみでありかつ対象プログラム中で一度も使用されない 1 個のメソッドは，未実装と判断し対象外とした．String 型の結合演算を行う 2 個のメソッドと Date 型の演算を行う 1 個のメソッドは，Alloy でそれらの演算に対応していないため記述できず，対象外とした．

4.3 結果

実験の結果，本手法で発見・検出できた表明に関する誤りは表 4 のようになった．ここで個数は，同じ場所に付与されている同じ種類の表明をまとめて 1ヶ所と数えた．例えば，同じクラスに付与されている不変条件は，複数あっても 1ヶ所となる．あるいは，同じメソッドに複数の事前条件と複数の事後条件が付与されている場合は事前条件が 1ヶ所，事後条件が 1ヶ所の計 2ヶ所となる．

表明の洗練が不十分なものは不変条件が 1ヶ所，事後条件が 1ヶ所であった．実装と表明との不一致については，事前条件が 1ヶ所，事後条件が 4ヶ所であった．それぞれについて次小節以降で詳細に述べる．

充足例・反例が Alloy Analyzer によって何番目に出力されたものかを以降で記しているが，これは実行環境に依存するものである．

4.3.1 表明の洗練が不十分

1. ContainerItem クラスの getAmount() の事後条件
   これは 3.1.1 節の「事後条件が不十分であり，メソッドのふるまいを十分に表現できていない」に該当する．

   コンテナはフィールドとして格納品目リストを所持している．getAmount() には戻り値に関する事後条件として,「戻り値と等しい数量をもつ品目が格納品目リスト内にが存在する，あるいは戻り値は 0 である」が定義されている．図 15 は Alloy Analyzer が getAmount() の表明の充足例として 2 番目に出力したものである．出力された充足例では，戻り値 (図中の$check_getAmount_org_a)6 と等しい数量をもつ品目 (Item0, Item2) が格納品目リスト ($check_getAmount_org_c の itemlist) に含まれている．しかし，引数 ($check_getAmount_org_n) は String2 であり，これは Item0 の品名 (String1) でも Item2 の品名 (String2) でもなく，またこのコンテナ ($check_getAmount_org_c) が格納品目リストとして所持している品目には品名が引数と同じものは無い．よって，この場合は戻り値として 0 を返すような表明に変更するべきだと考えられる．

2. Storage クラスの不変条件

表 4: 本手法で発見・検出された表明に関する欠陥と種類

表明の種類	対象とした表明の総数	表明の洗練が不十分	実装と表明との不一致
不変条件	6	1	0
事前条件	21	0	1
事後条件	23	1	4
合計	50	2	5

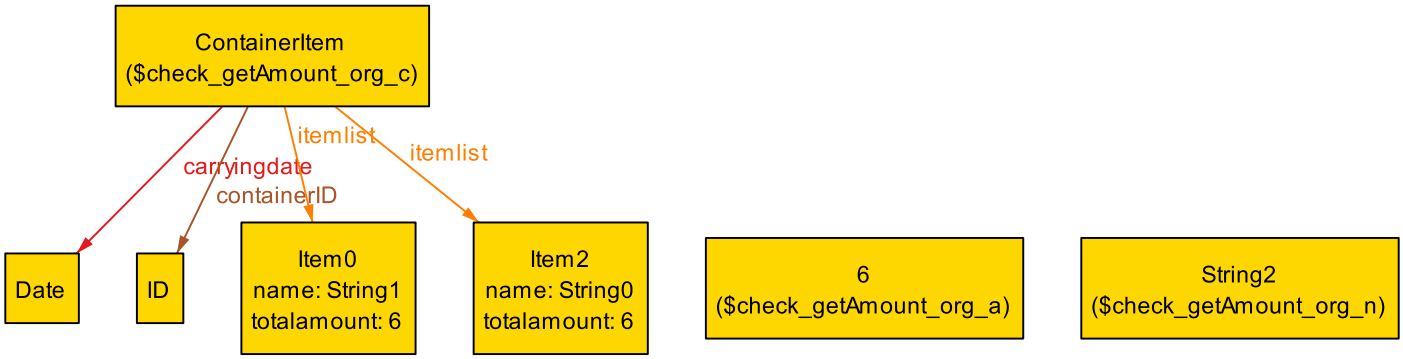


図 15: 充足例 (getAmount())

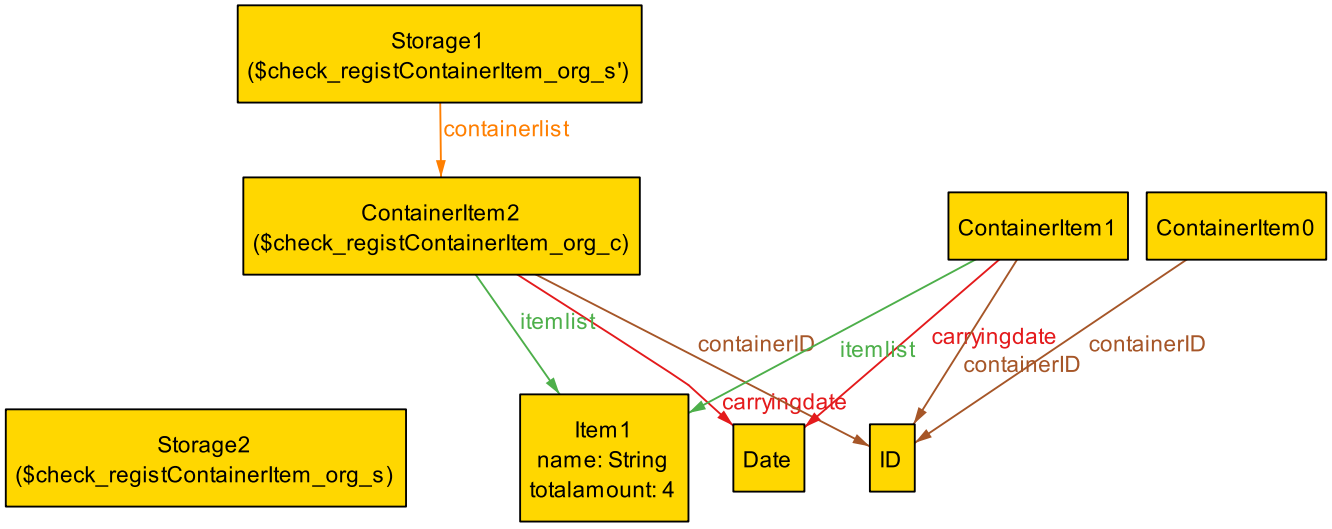


図 16: 充足例 (registContainerItem())

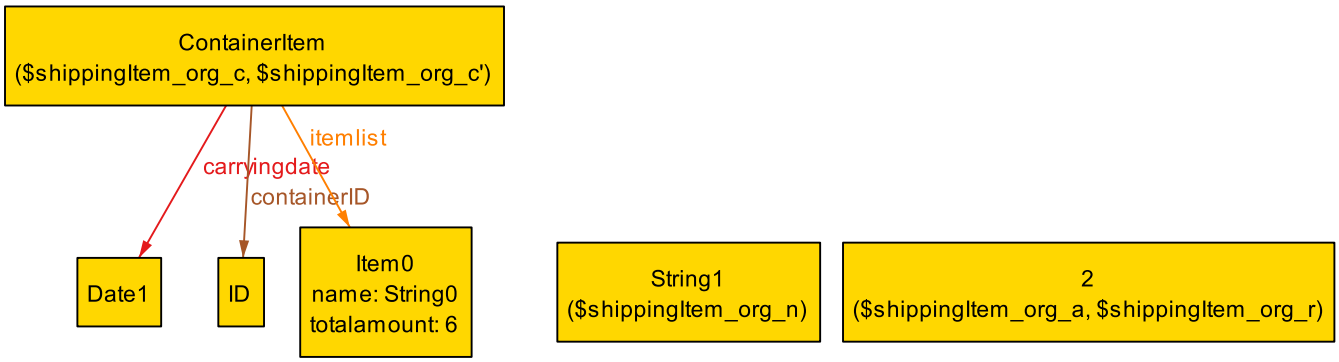


図 17: 反例 (shippingItem())

これは 3.1.1 節の「不変条件が不十分であり，クラスの性質を十分に表現できていない」に該当する．

倉庫はフィールドとして，コンテナのリストとそれらの合計を表す在庫リストを所持している．また倉庫には不変条件として,「各フィールドは null ではない」が定義されている．図 16 は Alloy Analyzer が倉庫のメソッド registContainerItem() の表明の充足例として 1 番目に出力したものである．メソッドの実行後の倉庫 ($check_registContainerItem_org_s') を見ると，コンテナリスト (containerlist) を所持しているが，在庫リスト (allitemlist) を所持していない．よって,「在庫リストはコンテナリストが所持している品目の合計である」という表明を追加するべきだと考えられる．

4.3.2 実装と表明との不一致

1. ContainerItem クラスの shippingItem() の事後条件
   これは 3.1.2 節の「事後条件が誤っており，メソッドのふるまいと一致しない」に該当する．

   コンテナはフィールドとして格納品目リストを所持している．shippingItem() には戻り値に関する事後条件として,「引数の文字列と品名が等しい品目が格納品目リスト内にあり，かつ戻り値は引数の整数からその品目の数量を引いたものである」が定義されている．shippingItem() の実装がこの表明を満たさない例として Alloy Analyzer が 1 番目に出力したものが図 17 である．これは，引数の文字列 ($shippingItem_org_n) と品名が等しい品目が格納品目リスト (itemlist) 内に存在しない例である．この状況は十分に起こりうるため，引数の文字列と品名が等しい品目が格納品目リスト内に存在する場合と存在しない場合で場合分けをした事後条件に修正する必要があると考えられる．

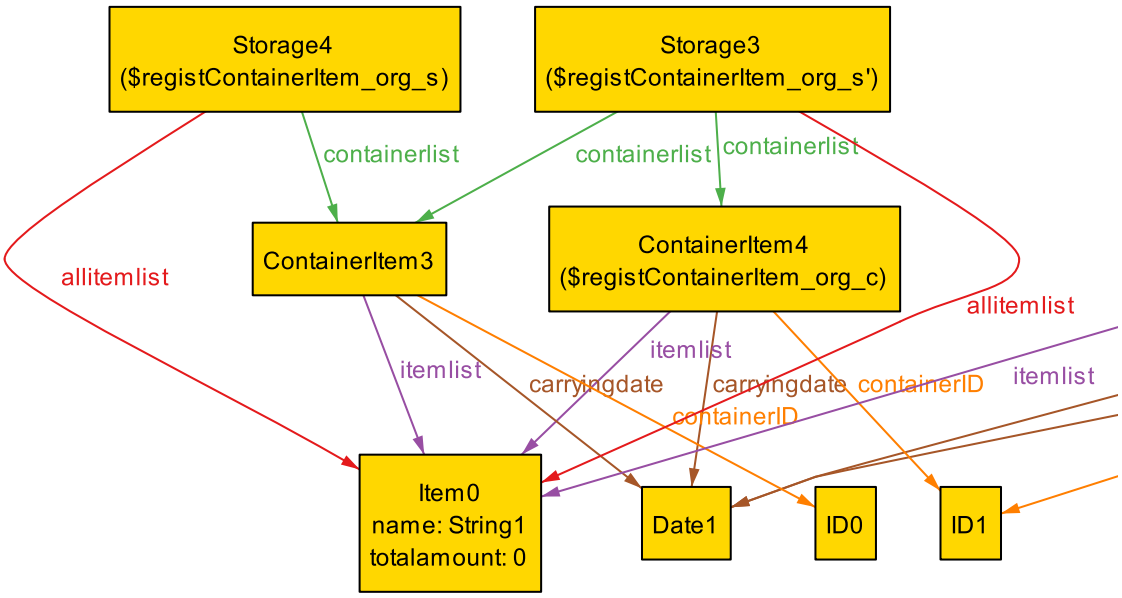


図 18: 反例 (registContainerItem()(updateAllItemList()))

2. Storage クラスの registContainerItem() の事後条件

3. Storage クラスの updateAllItemList() の事後条件
   これらは 3.1.2 節の「事後条件が誤っており，メソッドのふるまいと一致しない」に該当する．

   updateAllItemList() は registContainerItem() 中で呼び出されるプライベートメソッドである．倉庫はフィールドとして，在庫リストを所持している．この 2 つのメソッドには共通の事後条件として,「実行前の在庫リストが空でなければ，実行後の在庫リストと引数のコンテナの格納品目リストは異なる」が定義されている．registContainerItem() の実装がこの表明を満たさない例として Alloy Analyzer が 1 番目に出力したグラフの必要な部分を切り出したものが図 18 である．updateAllItemList() でも同様の反例が出力される．これは，実行前の倉庫 ($registContainerItem_org_s) のコンテナリスト (containerlist) 内のすべてのコンテナの格納品目リスト内のすべての品目の品目数が 0，引数のコンテナ ($registContainerItem_org_c) の格納品目リスト内のすべての品目の品目数が 0 となっており，表明を満たさない．修正案として，この事後条件を削除する，あるいは実行前の全品目のリストに含まれるすべての品目の数量が 0 である場合を考慮した事後条件に修正するといった方法が考えられる．

   あるいは,「引数のコンテナの格納品目リスト内の品目には品目数が 0 でないものが存

在する」という事前条件を追加することでこの状態を除外するという修正案も考えられる．この場合は,「不変条件・事前条件が不十分であり，実装が事後条件・不変条件を満たさない」に該当する．

4. ReceiptionDesk クラスの receiptRequest() の事後条件
   これは 3.1.2 節の「事後条件が誤っており，メソッドのふるまいと一致しない」に該当する．

   受付係はフィールドとして，倉庫を所持している．receiptRequest() には戻り値に関する事後条件として,「戻り値は，実行後の倉庫が引数の注文を満たせるかどうかである」が定義されている．receiptRequest() の実装がこの表明を満たさない例として Alloy Analyzer が 1 番目に出力したグラフの必要な部分を切り出したものが図 19 である．これは，実行前の倉庫 (Storage0) がある品名 (String1) の品目を 6 つ所持し，引数としてその品名の品目を 4 つ要求する注文 ($receiptRequest_org_r) が指定された場合に戻り値 ($receiptRequest_org_rslt) が真となる例である．このとき，実行後の倉庫は引数の注文を満たせないが，実装では戻り値は引数の注文に対応できたかどうかを表す．したがって,「戻り値は，実行前の倉庫係が引数の注文を満たせるかどうかである」という事後条件に修正する必要があると考えられる．

5. ReceiptionDesk クラスの receiptNewContainerItem() の事前条件
   これは 3.1.2 節の「不変条件・事前条件が不十分であり，実装が事後条件・不変条件を満たさない」に該当する．

   倉庫には，不変条件として,「コンテナリスト内で ID は重複しない」が定義されている．receiptNewContainerItem() の実装がこの表明を満たさない例として Alloy Analyzer が 1 番目に出力したグラフの必要な部分を切り出したものが図 20 である．これは，実行前の倉庫 (Storage0) が所持しているコンテナ (ContainerItem1) の ID と引数のコンテナ ($receiptNewContainerItem_c) の ID が同じものである例である．解決策として,「引数のコンテナの ID は，倉庫係が所持しているコンテナと重複しない」という事前条件が必要だと考えられる．

6. ReceiptionDesk クラスの deliveringOrder() の事後条件
   これは 3.1.2 節の「事後条件が誤っており，メソッドのふるまいと一致しない」に該当する．

   受付係はフィールドとして，不足注文リストを所持している．deliveringOrder() には実行後の状態に関する事後条件として,「戻り値に含まれているある品名の品目の合計と数量と品名が等しい注文が実行後の注文リスト内にあれば，その注文は配送済みである」

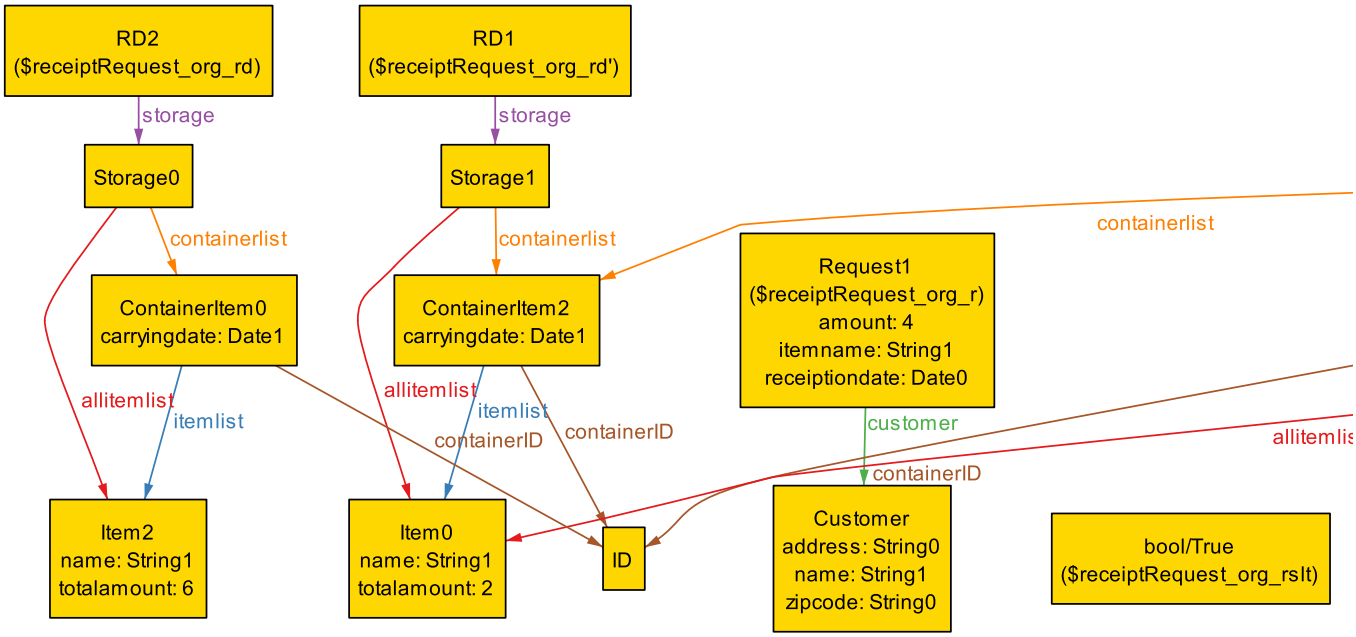


図 19: 反例 (receiptRequest())

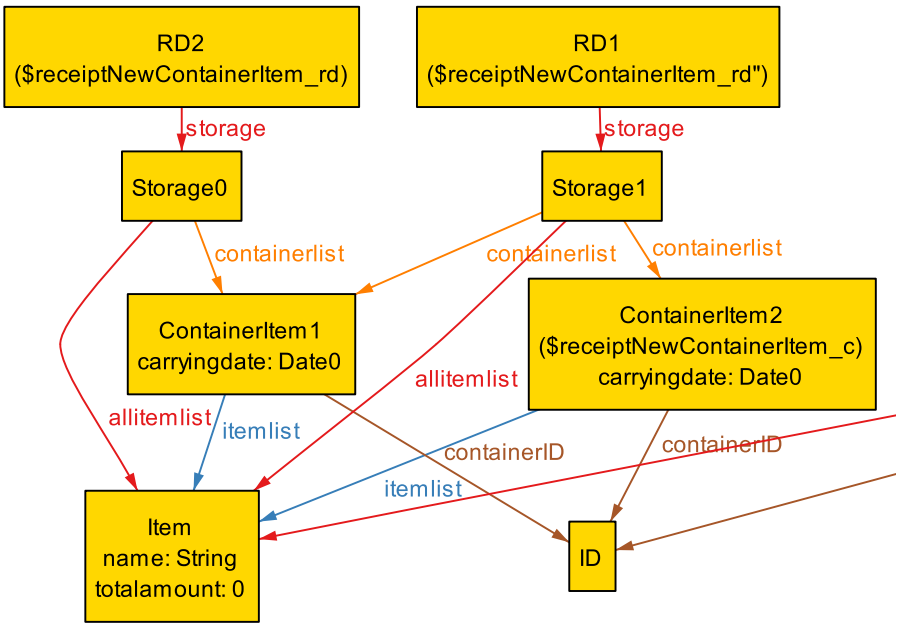


図 20: 反例 (receiptNewContainerItem())

が定義されている．deliveringOrder() の実装がこの表明を満たさない例として Alloy Analyzer が 1 番目に出力したグラフの必要な部分を切り出したものが図 21 である．これは，実行前の受付係 (RD2) は数量・品名ともに等しくかつ未配送 (SHORTAGE) である 2 つの注文 (Request1 と Request2) を所持しており，そしてメソッドの実行によりそのうちの片方 (Request1) が配送された．その結果，実行後の受付係 (RD1) では配送された注文は (Request1) は配送済み (DELIVERED) の注文 (Request0) となっており，もう一方の注文 (Request2) は未配送のままである，という例である．

deliveringOrder() の戻り値は，実行前の倉庫係 (Storage0) が所持しているコンテナと実行後の倉庫係 (Storage1) が所持しているコンテナの内容の差分を表すコンテナのリストであり，ここでは ContainerItem0 が戻り値となる．このとき，実行後の注文リストに含まれる 2 つの注文 (Request0 と Request2) はともに,「戻り値に含まれているある品名の品目の合計と数量と品名が等しい注文」であるが，実行前には 1 つの注文に対応する品目しかなかったため，Request2 はこの事後条件を満たさない．これは，この事後条件が同じ品名・数量の注文が注文リストに含まれる状態を考慮していないためと考えられる．したがって，この状態も考慮した事後条件に修正する必要があると考えられる．

4.4 考察

4.4.1 対応できなかった点

本手法では発見・検出できなかったが，目視による確認で発見した表明に関する欠陥を表 5 に示す．

これは 3.3.1 節で述べた，JML から Alloy へ変換できなかった部分のために本手法では検出できなかった．表 5 の事後条件の洗練が不十分なものに対応できなかったのは，文字列型の参照の比較を扱えなかったためである．これは実験の対象とした 50ヶ所の表明の中の 1ヶ所が該当した．解決策として，文字列型を参照と値の 2 つのフィールドを持つシグネチャとして定義するものが知られている [27]．本手法では，記述が複雑となってしまうため採用しなかったが，今後対応することを検討している．

表 5 の不変条件の実装との不一致に対応できなかったのは，JML の \typeof() と \type() を Alloy で記述することができなかったためである．これは実験の対象とした 50ヶ所の表明の中の 16ヶ所が該当した．解決策として，シグネチャのフィールドとしてリストを所持するときに，特定の型の集合としてではなく，全型の集合として記述することが考えられる．しかし，これでは記述が煩雑となってしまい，可読性も損なわれる．また，実装として総称

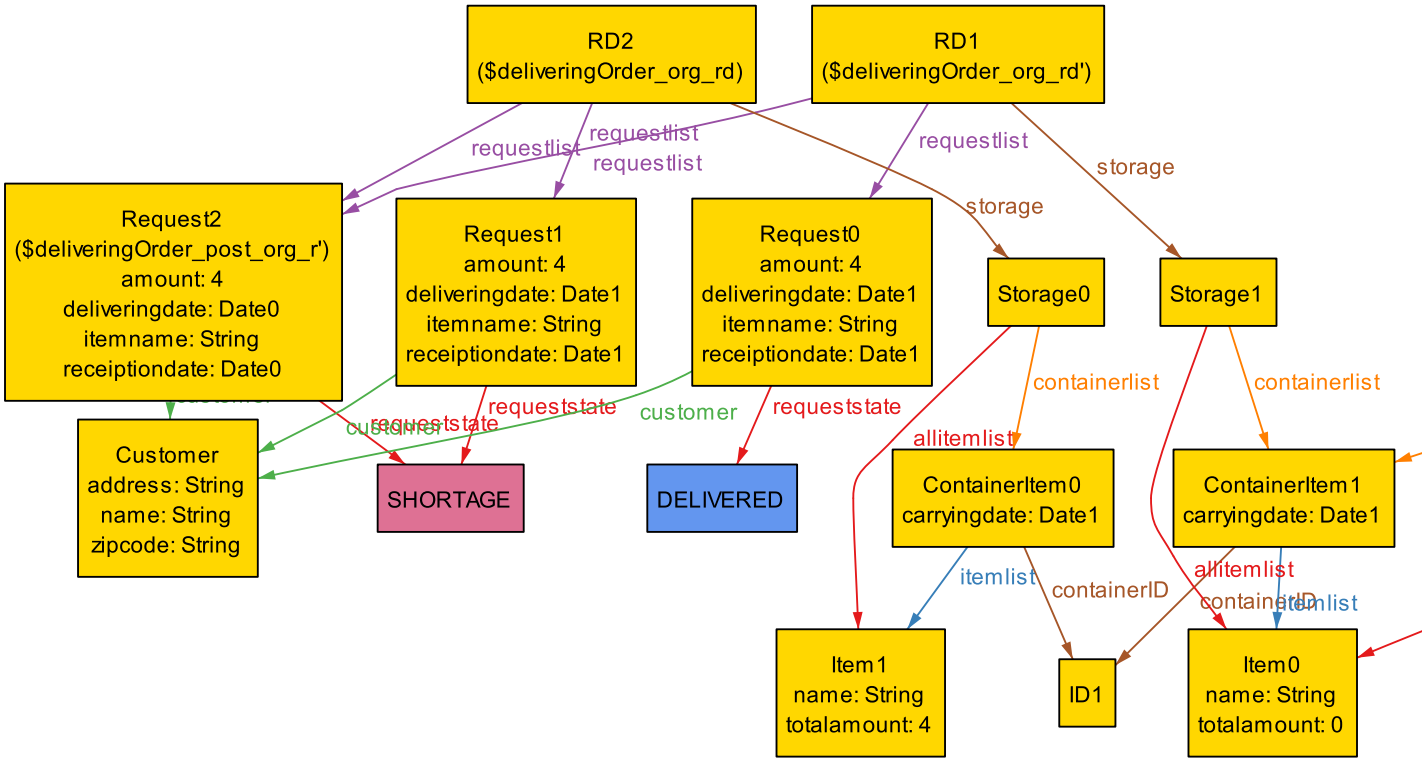


図 21: 反例 (deliveringOrder())

表 5: 本手法で発見・検出できなかった表明に関する欠陥と種類

表明の種類	表明の洗練が不十分	実装と表明との不一致
不変条件	0	1
事前条件	0	0
事後条件	1	0
合計	1	1

型を使用していれば，コンパイラの型チェック機能で検出できると考えられるので対応する必要性は低いと考え，本手法では対応しなかった．

また同じく 3.3.1 節で述べた，JML から Alloy へ変換できなかった部分として他に集合型の null を扱えなかったために記述できなかったものがあり，実験の対象とした 50ヶ所の表明の中の 3ヶ所が該当した．

3.3.2 節で述べた，Java から Alloy へ変換できなかったループに関しては，実験の対象とした 50ヶ所の表明の中の 3ヶ所が該当した．しかし，代用とした記述によりこのうち 2ヶ所で実装と使用との不一致を検出できたことから，代用とした記述による検証でも有用であると考えられる．

またここまでで挙げたもの以外に，Alloy Analyzer の「集合の集合」の解析に対する制限のために，実装と表明との不一致の検査ができなかったものが存在した．実験の対象とした 22 のメソッドのうちの ReceiptionDesk クラスの deliveringOrder() がこれに該当した．このメソッドは戻り値がコンテナのリストであり，またコンテナは所持品目リストを所持しているため集合の集合となってしまう．対応策として，次の 2 つを行った．

- 引数のコンテナリストがただ一つのコンテナのみを含む場合に限定して解析を行う
- 引数なしのメソッドとして記述し，引数以外に関する表明のみを検査する

1 つ目の対応策は，反例が検出されなかったとしても実装と表明との不一致が存在しないことを示すことはできない．しかし，この解析によって反例が検出された場合は実装と表明との不一致が存在することを示せる．2 つ目の対応策は，引数に関する表明について示すことはできないが，それ以外の表明については，解析の制限に該当しないメソッドと同じ検査を行うことができる．このような 2 つの対応策により，検査を行える範囲を広げた．そして前小節で述べたように，ReceiptionDesk クラスの deliveringOrder() の「事後条件が誤っており，メソッドのふるまいと一致しない」という実装と表明との不一致を検出することができた．

4.4.2 記述コスト

本手法のために記述した Alloy の行数を表 6 に示す．Alloy のファイルの単位はモジュールとなる．ElementType は，各モジュールで共通して使用される String シグネチャや Date シグネチャの宣言を行っている．また，StockState クラスで宣言されている注文の状態を表す列挙型の宣言は Request にまとめた．

表 3 と比較すると，コンストラクタを合わせたメソッドの合計 52 のうち 23 しか対象としていないにも関わらず，総行数は約 1.5 倍に増加している．項目ごとに見ていくと，JML か

表 6: 発見・検出のために記述した Alloy の行数 (空行・コメント行を含む)

モジュール名	JML から変換	Java から変換	その他	総行数
ElementType	0	0	4	4
Customer	48	30	119	197
Request	32	41	97	170
Item	15	12	55	82
ContainerItem	29	41	82	152
Storage	60	104	183	347
ReceiptionDesk	37	177	151	365
合計	221	405	691	1317

ら変換した部分に関してはほぼ同程度だと考えられる．これは 3.3.1 で述べたように，JML の構文と Alloy の構文はほぼ対応するためである．また，この部分は変換の自動化が比較的容易だと考えられる．Java から変換した部分に関しては，実際に対象としたメソッドが約半数であることを考えると大幅に増加していると考えられる．これは 3.3.2 節で述べたように，Java では陰に示される条件を Alloy では陽に記述しなければならないためである．これを完全に自動変換することは困難であると考えられるが，変化する部分 (Java で陽に記述されている部分) の Alloy 記述を入力とし，変化しない部分 (Java では陰に示される部分) の Alloy 記述を補ったものを出力するツールは実現可能だと考えている．その他の部分は合計 691 と全体の約半数を占めているが，これは主に，3.2 節で述べた「表明同士の関係」の記述や「実装と表明の関係」の記述および Alloy Analyzer に解析を指示する命令であり，メソッド名と引数と戻り値の型および副作用の有無を与えてやれば自動的に生成できるものである．

本研究では変換を手動で行ったが自動化できる部分も多く，変換ツールを実装することで記述コストを大幅に下げることができる．また,「表明同士の関係」の記述や「実装と表明の関係」の記述は直接ツールの使用者に見せる必要の無い部分だと考えられるので，ツールの実装の際にはそれらの部分を使用者に隠蔽するようにすれば可読性が向上し，使用者が補わなければならない記述部分に注視させることができるのではないかと考えている．

5　あとがき

本研究では，JML で記述された表明の洗練の程度を可視化することで洗練の不十分さを使用者が発見しやすくする手法，および JML で記述された表明と Java の実装との不一致を検出する手法を提案した．

提案手法では，JML で記述された表明と Java で記述された実装を Alloy に手動で変換し，その Alloy 記述を使用して，メソッドの表明がすべて満たされたことを表す論理式とメソッドの実装が表明を満たさないことを表す論理式を記述し，Alloy Analyzer に入力することで充足例や反例をでグラフとして出力する．また，在庫管理プログラムに対する適用実験を行った．

結果として，ESC/Java2 だけでは見つけられなかった 6ヶ所の欠陥を検出することができ，また 2ヶ所の洗練の不十分さを指摘することができた．この結果より，提案手法の有効性を確認できた．また，複数の手法を使用することで互いに欠点を補いあい，より信頼性の高い設計・実装が行えることがわかった．

この手法の欠点として，作業，特に手動で行っている変換にかかるコストが大きいことが挙げられる．よって今後の課題としては，変換を支援するようなツールを考案・開発することが考えられる．例えば，部分的な自動化を行うツールや，必要な記述を補うツールが考えられる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心より感謝申し上げます．

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 岡野 浩三 准教授に深く感謝申し上げます．

本研究に関して，的確なご助言ご指導を頂きました 井垣 宏 特任准教授に心より感謝申し上げます．

本研究を行うにあたり，日常の議論の中でご助言を頂きました 肥後 芳樹 助教に心より感謝申し上げます．

本報告を行うにあたりご指導，ご協力を頂き，さらに日常でも声をかけて頂きました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程２年の 花田 健太郎 氏，同 吉岡 一樹 氏に深く感謝申し上げます．

また研究室生活の中で相談に乗って頂き，また励まして頂きました 大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程１年の 榛葉 浩章 氏，同 佐々木 幸広 氏に心より感謝申し上げます．

その他の楠本研究室の皆様のご協力に心より感謝致します．

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[1] Meyer, B.: *Object-oriented software construction (2nd ed.)*, Prentice-Hall, Inc. (1997).

[2] Back, R. and von Wright, J.: Refinement calculus: a systematic introduction, *Graduate Texts in Computer Science*, Springer (1998).

[3] Leavens, G., Baker, A. and Ruby, C.: JML: A notation for detailed design, *in Behavioral Specifications of Businesses and Systrems*, pp. 175–188 (1999).

[4] Cok, D. and Kiniry, J.: Esc/java2: Uniting esc/java and jml, *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, Springer, pp. 108–128 (Mar. 2004).

[5] 中島震：プログラム簡易検証ツール ESC/Java2，コンピュータ ソフトウェア，Vol. 24, No. 2, pp. 2–7 (2007).

[6] Jackson, D.: Alloy: a lightweight object modelling notation, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 11, No. 2, pp. 256–290 (2002).

[7] Jackson, D.: *Software abstractions: logic, language, and analysis*, MIT Press (MA), revised edition (2011).

[8] 中島震，鵜林尚靖：Alloy：自動解析可能なモデル規範形式仕様言語，コンピュータソフトウェア， Vol. 26, No. 3, pp. 78–83 (2009).

[9] 尾鷲方志，岡野浩三，楠本真二：在庫管理プログラムの設計に対する JML 記述と ESC/Java2 を用いた検証の事例報告 (研究速報)，電子情報通信学会論文誌，Vol. J91D, No. 11, pp. 2719–2720 (2008).

[10] 森恵弥佳，岡野浩三，楠本真二：在庫管理プログラムに対する Alloy Analyzer を用いた検証事例，ウィンターワークショップ 2013・イン・那須論文集，pp. 5–6 (Jan. 2013).

[11] Jackson, D.: Alloy: a language & tool for relational models, `http://alloy.mit.edu`.

[12] 梅村晃広：SAT ソルバ・SMT ソルバの技術と応用，コンピュータ ソフトウェア，Vol. 27, No. 3, pp. 24–35 (2010).

[13] JML: Reference Manual, `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`.

[14] Khalek, S., Yang, G., Zhang, L., Marinov, D. and Khurshid, S.: TestEra: A tool for testing Java programs using alloy specifications, *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 608–611 (Nov. 2011).

[15] Hansen, K. and Ingstrup, M.: Modeling and analyzing architectural change with alloy, *Proceedings of the 2010 ACM Symposium on Applied Computing(SAC)*, pp. 2257–2264 (Mar. 2010).

[16] 小飼敬，上田賀一，大久保訓，高橋勇喜，中野利彦：Alloy を利用した情報制御システム記述言語の仕様検証の実用化，コンピュータソフトウェア， Vol. 27, No. 4, pp. 228–233 (2010).

[17] Ganov, S., Khurshid, S. and Perry, D.: Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science, Vol. 7635, Springer, pp. 414–429 (2012).

[18] Near, J. and Jackson, D.: An imperative extension to Alloy, *Proceedings of the Second international conference on Abstract State Machines, Alloy, B and Z*, Springer, pp. 118–131 (Feb. 2010).

[19] Giannakopoulos, T., Dougherty, D., Fisler, K. and Krishnamurthi, S.: Towards an operational semantics for alloy, *Proceedings of the 2nd World Congress on Formal Methods*, Springer, pp. 483–498 (Nov. 2010).

[20] Anastasakis, K., Bordbar, B., Georg, G. and Ray, I.: On challenges of model transformation from UML to Alloy, *Software and Systems Modeling*, Vol. 9, No. 1, pp. 69–86 (2010).

[21] Garis, A., Cunha, A. and Riesco, D.: Translating alloy specifications to UML class diagrams annotated with OCL, *Proceedings of the 9th international conference on Software Engineering and Formal Methods (SEFM)*, Lecture Notes in Computer Science, Vol. 7041, Springer, pp. 221–236 (Nov. 2011).

[22] Salay, R., Famelis, M. and Chechik, M.: Language Independent Refinement Using Partial Modeling, *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Vol. 7212, Springer, pp. 224–239 (Mar. 2012).

[23] 酒井政裕，今井健男，片岡欣夫：データ構造に関する仕様を含め検証できる C 言語プログラム部品検証ツール CForge (特集製品ライフサイクルでの高信頼化技術)，東芝レビュー， Vol. 64, No. 8, pp. 20–23 (2009).

[24] Clarke, E., Kroening, D. and Lerda, F.: A tool for checking ANSI-C programs, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, Vol. 2988, Springer, pp. 168–176 (Mar./Apr. 2004).

[25] Barnett, M., Leino, K. and Schulte, W.: The Spec# programming system: An overview, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, Lecture Notes in Computer Science, Vol. 3362, Springer, pp. 49–69 (Mar. 2004).

[26] Hoare, C.: Proof of correctness of data representations, *Acta informatica*, Vol. 1, No. 4, pp. 271–281 (1972).

[27] Rupakheti, C. and H., D.: Finding errors from reverse-engineered equality models using a constraint solver, *28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 77–86 (Sep. 2012).