Alloy Analyzer によるモデリング入門 チュートリアル

小林健一

2013 年 1 月 18 日

1　初めに

ソフトウェアの欠陥がもたらす影響がクリティカルになるにつれて、より間違いの少ない開発手法に注目が集まっています。

そんな中で注目されている技術の一つが形式手法です。形式手法とは、数学をベースにしたシステム開発手法の総称です。

ですが、従来からの形式手法は、数学の専門知識が必要であったり、大がかりなシステム開発体制と合わせて語られることがほとんどでした。このため、一般の開発者にとってはなじみのない技術でした。

そんな状況が、ここ数年で一変しました。PC の性能向上と、より使いやすいツールの提供により、誰でも簡単に試すことのできる環境が整いつつあります。

本稿では、形式手法ツールの一つ、Alloy Analyzer を取り上げ、以下の 2 点を中心に解説します。

- 形式手法の導入の難しさは解消されてきている。
- 形式手法は他の現行の開発手法を補う形で利用可能である。

1.1　なぜ Alloy Analyzer が開発されたのか?

みなさんが「ソフトウェアの欠陥を除去する」ことを考えるとしたら、まず考えるのはソフトウェアテストを行なうことでしょう。ですがご存じの通り、テストではソフトウェアの欠陥を完全に排除することはできません。

> 「テストでは、欠陥が無いことは保証できない」(Dijkstra)

テストは特定の条件を入力としたときの結果を確認します。ですので、テストケースには抜け漏れの危険がつきまといます。この問題を解決するため、形式手法では仕様を数学的に記述し、それを証明しようとします。この証明を人力で行うのは限界があるので、自動化技術として、自動定理証明やモデル検査があります。これらの技術は自動で全ての条件や状態を洗い出すので、テストと比較すると完全性が保証されるというメリットがあります。

ですが、こういった自動化技術にも弱点があります。例えば、モデル検査では与えられたモデルを基に、全ての状態を網羅するため、膨大なコストがかかるのです。この状態爆発という現象は、最新の計算環境をもってしても解決が難しい問題です。

ここで、Alloy Analyzer の開発者である Daniel Jackson は一つの仮説を打ち出しました。

Small Scope Hypothesis

「ほとんどの欠陥は、小さい探索範囲であっても反例として発見される」

全てを網羅的に検証するのでなく、最も欠陥が発生しやすいところを集中的に検証しようということです。Daniel Jackson は、この考え方に則った形式手法を、「Lightweight Formal Methods」と呼んでいます。

完全な証明よりも、現実的なコストの範囲内での欠陥の発見を目指すのです。Alloy Analyzer は、Lightweight Formal Methods の考え方に基づいた形式手法ツールの一つです。

1.2 どういうことができるのか?

Alloy Analyzer は以下の特徴を持っています。

- 集合論と述語論理を理論基盤に持つ。
- 与えられた仕様に対し、指定したプロパティを満たす、例を探すことができる (述語の検証)。
- 与えられた仕様に対し、指定したプロパティを満たさない、反例を探すことができる (表明の検証)。

特に Alloy Analyzer を使って設計を行なう際、上記の例・反例の探索が非常に役立ちます。通常、クラス図やデータモデルを作成した際、レビューは手作業で行なうことが多いかと思います。Alloy Analyzer では手軽にデータ構造を表す仕様を記述できるため、クラス図の多重度の間違いや、複数の制約条件の組合わせの結果を、その場で確認することができます。

2 Alloy Analyzer によるアジャイルモデリング

2.1 Alloy Analyzer のインストール

Alloy Analyzer を動作させるには、以下の環境が必要です。

- JDK6.0 以上
- Alloy Analyzer `http://alloy.mit.edu/alloy/download.html`

```
> java −jar alloy4.2.jar
```

とすることで、Alloy Analyzer が起動します。

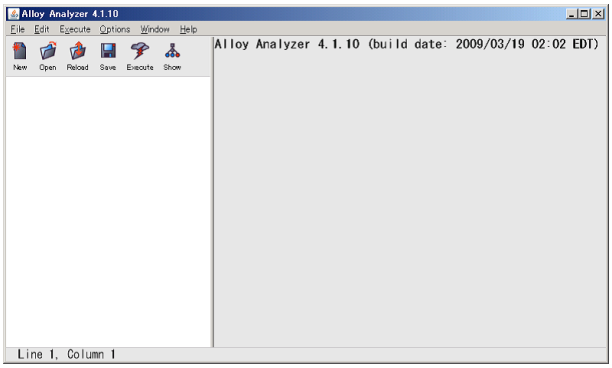

図 1: Alloy Analyzer 起動の様子

2.2 Alloy Analyzer の基礎用語

Alloy Analyzer を使用する上で、いくつか用語の説明をしておきます。

モデル (Model) ソフトウェアの抽象化された記述。

仕様 (Specification) Alloy Analyzer で記述されたモデル。.als ファイル。

アトム (Atom) 集合の要素。プリミティブな要素を表します。

関係 (Relation) 要素を関係づける構造。

インスタンス (Instance) Alloy Analyzer が出力する例・反例。

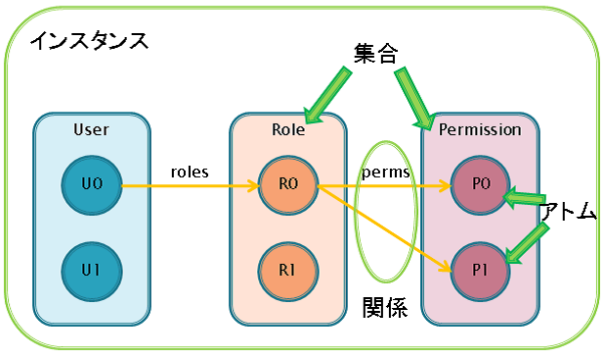

図 2: Alloy Analyzer の基礎用語

インスタンスは仕様を検証した結果です。インスタンスの中で、アトム同士の関係を調べることで、自分たちが期待した結果が出ているかを確認することができます。

以下の節では、Alloy Analyzer を使い、モデリングの様子を体験します。

2.3 モデリングセッション 1:最小限の仕様を作る

今回モデリングする対象は、以下の簡単なロールベースアクセス制御 (Role-Based Access Control:RBAC) の仕様です。

- 集合

  ユーザ (User) ユーザアカウント。

  ロール (Role) ユーザに割り当てられる役割。Administrator、Guest など。

  権限 (Permission) 役割に割り当てられる権限。Read,Write,ReadWrite など。

- ユーザは、0 以上のロールを持つ。
- ロールは、0 以上の権限を持つ。

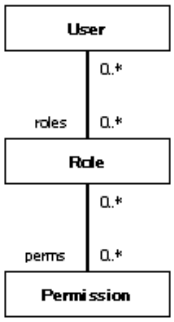

図 3: RBAC のクラス図

Alloy Analyzer では、最初から一気に仕様を書き上げることはあまり行ないません。まずは最小限の仕様を作成したうえで、検証を繰り返しながら仕様を追加していきます。

2.3.1 集合の記述

最初はまず、最低限の仕様を作りましょう。

```
1 /* ユーザ */
2 sig User {}
3 /* ロール */
4 sig Role {}
5 /* 権限 */
6 sig Permission {}
7
8 run {}
```

これだけです。

ここでいくつか Alloy Analyzer のキーワードが出てきました。

- **sig**： Signature の略です。集合を表します。**sig** User {} と記述すると、User という名前の集合を記述したことになります。{} の中には他の集合との関係を記述します。
- **run**： 述語 (**pred**) を実行するコマンドです。{} の中に、例を出力する条件を記述します。

ここで実行 (Execute ボタン。Ctrl+E でも可) をすると、検証が実行されます。右側の結果より、「Instance」のリンクをクリックすることでインスタンスを確認できます。もし、アトムが 1 つも表示されない場合は、次のインスタンスを表示しましょう。「Next」ボタンにより、次のインスタンスを表示することができます。

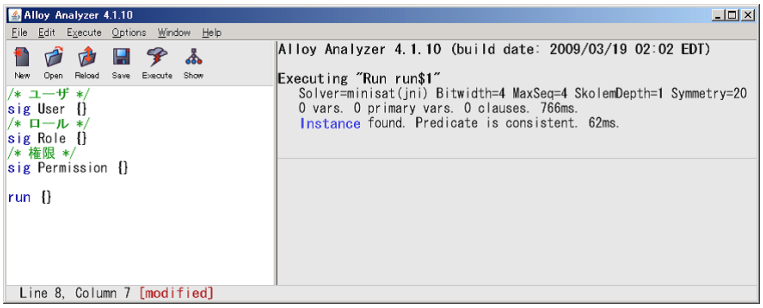

図 4: 実行

ここで Alloy Analyzer の強力な機能である、「Evaluator」を試しましょう。「Evaluator」ボタンを押すことで、Evaluator を開くことができます。

Evaluator では、任意の Alloy 式を記述できるので、インスタンスの内容を詳細に確認できます。

試しに User と記述すると、User のアトムが出力されます。User + Role と記述すると、User と Role の和集合が表示されます。

主な集合演算には以下の種類があります。

- A + B： 和集合
- A & B： 積集合
- A − B： 差集合

具体的に入力して確認してみましょう。

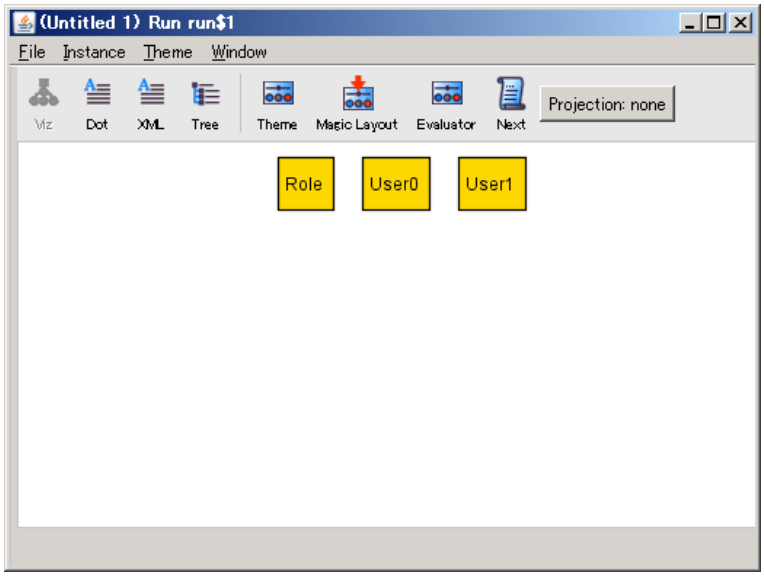

図 5: インスタンスの表示

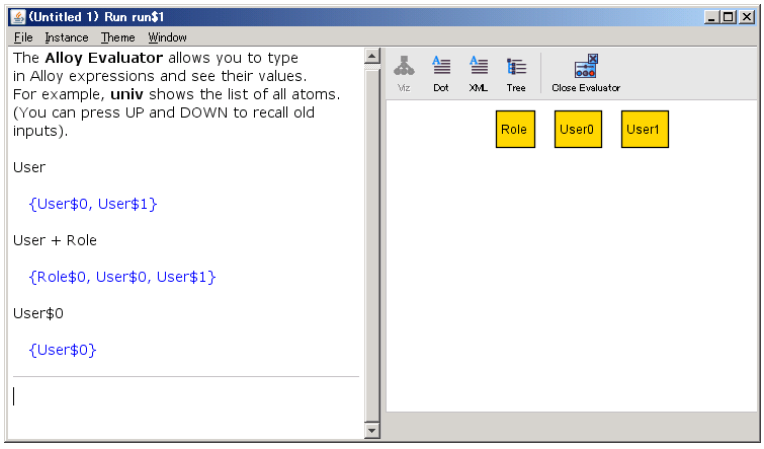

図 6: Evaluator

2.3.2 関係の記述

先ほどの仕様には関係が記述されていないので、各アトムには繋がりがありませんでした。次は関係を記述することで、アトム同士の繋がりを表現します。

```
1  /* ユーザアカウント
2      0 以上のロールを roles として持つ
3  */
4  sig User {
5    roles: set Role
6  }
7  /* ロール
8      0 以上の権限を持つ
9  */
10 sig Role {
11   perms: set Permission
12 }
13 /* 権限 */
14 sig Permission {}
15
16 run {}
```

関係は上記仕様のように、シグニチャの {} の中に記述します。
代表的な記述例は以下の通りです。

- **lone**： 0 または 1
- **one**： 1
- **set**： 0 以上
- **some**： 1 以上

関係は集合をつなげます。これは以下の様にテーブルとして表現することができます (以下は Alloy Analyzer の仕様ではありません。説明用の記述です)。

```
User = {(U0)}
Role = {(R0),(R1),(R2)}
Permission = {(P0),(P1),(P2)}
roles = {(U0,R0),(U0,R1)}
perms = {(R0,P2),(R1,P1),(R2,P0),(R2,P2)}
```

また実行してインスタンスを表示してみましょう。
関係に対しては関係演算を行なうことができます。ここではいくつかの関係演算を紹介します。

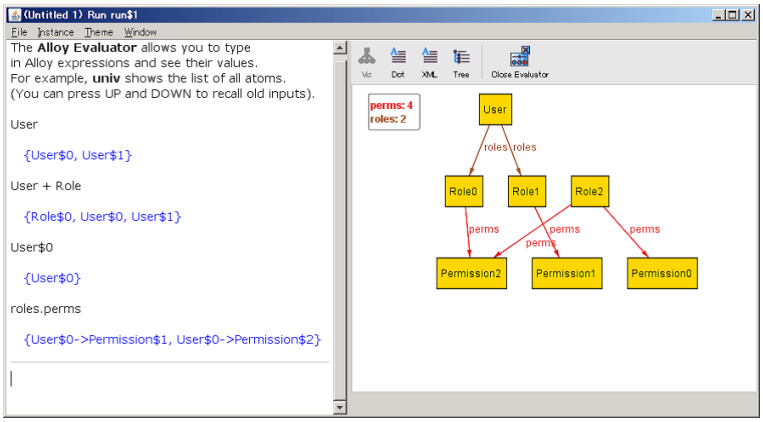

図 7: 関係の表示

- →： アロー。集合同士に対して使い、新しい関係を作り出す。
- .： ドットジョイン。2 つの集合をジョインする。

ここで、ドットジョインの動作には注意が必要です。ドットジョインは、ジョインの左右で対応する要素同士を消去する形で結びつけます。

$$\text{roles} = \{(U0,R0),(U0,R1)\}$$
$$\text{perms} = \{(R0,P2),(R1,P1),(R2,P0),(R2,P2)\}$$

上記の場合、roles.perms とすると、roles の右側の要素 R*と、perms の左側の要素 R^*のそれぞれ対応する要素が消去され、以下の集合が出来ることになります。

$$\text{roles.perms} = \{(U0,P1),(U0,P2)\}$$

この動作さえ押さえてしまえば、Alloy Analyzer の関係演算はマスターしたも同然です。

2.3.3 関数の定義

集合同士の関係が記述できたので、関係を辿って、「あるユーザの持つ権限の集合を取得する」関数を記述しましょう。
関数は以下のように記述します。

```
1 fun permissions[u:User]:set Permission {
2   u.roles.perms
3 }
```

- **fun**： 関数。関数名 [引数] :集合 {本体}の形式で記述する。

上記の例では、ある User のアトム u を引数として受取り、u の持つロールに紐付いた権限の集合を返すことになります。

この関数も Evaluator で直接試すことができます。User のアトム User$0 がインスタンスに存在するとして、以下の様に Evaluator で実行すると、結果の権限の集合が得られます。

```
permissions[User$0]
```

ここまでで完成した仕様を以下に示します。

```
/*
  Role Based Access Control その 1

  author: Kenichi Kobayashi

*/

/* ユーザ。
   0 以上のロールを roles として持つ
*/
sig User {
  roles: set Role
}

/* ロール。Admin,Guest など。
   0 以上の権限を持つ
*/
sig Role {
  perms: set Permission
}

/* ロールに割り当てられる権限。Read,Write,ReadWrite など
*/
sig Permission {}

/* あるユーザの持つ権限の集合を取得する */
fun permissions[u:User]:set Permission {
  u.roles.perms
}

/* run コマンドは pred(述語) を指定できる */
run {}
```

分からないところはどんどん Evaluator に入力して試すことができるので、簡単に実行できるということがお分かりいただけたと思います。

2.4 モデリングセッション 2:仕様を拡張する

先ほどの仕様はごく簡単に、最低限の機能を記述したものでした。次はこの仕様を拡張しましょう。

今回は、以下の仕様追加を行ないます。

- ロール同士に階層関係を持たせる。経理課職員、人事課職員は総務部職員をスーパーロールに持つなど。
- ユーザにロールを追加、削除する機能を追加する
- ロールに権限を追加、削除する機能を追加する
- ロールを追加して削除した場合、ユーザのロールは最初の状態に戻ることを検証する

2.4.1 ロールの階層関係

Role の階層関係を表現するには、以下の様に記述します。

```
1  sig Role {
2    perms: set Permission,
3    superRoles: set Role
4  }
```

Alloy Analyzer には汎化関係の記述もありますが、今回の様なケースでは汎化は使いません。

ただし、これだけではロールの階層構造がループしてしまい、経理課職員のスーパーロールが経理課職員、という状況が発生してしまいます。これを防ぐため、以下の記述を追加します。

```
1    fact {
2      no r:Role | r in r.^superRoles
3    }
```

- r:Role は Role のアトム r を表します。
- e **in** A と書いた場合、「e は A の要素である」ことを表します。
- **no** x:A | V(x) と書いた場合、通常の論理式では¬ (∃ x:A. V(x))、つまり V を満たす A の要素はないことを表わします。

- ^は推移閉包を表します。r.^superRolesは、rからsuperRolesを辿っていってたどり着ける全てのRoleの要素の集合を表します。

上記の記述は、ロールrからsuperRolesを辿っていった時、その中にr自身が含まれることはない、という不変制約を表しています。

2.4.2 追加、削除機能

ユーザへのロール追加削除、およびロールへの権限追加削除は以下の様に記述します。

```
1  /* ユーザにロールを付与する。*/
2  pred addRole[u,u':User,r:Role] {
3     u'.roles = u.roles + r
4  }
5
6  /* ユーザからロールを剥奪する。*/
7  pred removeRole[u,u':User,r:Role] {
8     u'.roles = u.roles - r
9  }
10
11 /* ロールに権限を付与する。*/
12 pred addPermission[r,r':Role,p:Permission] {
13    r'.perms = r.perms + p
14 }
15
16 /* ロールから権限を剥奪する。*/
17 pred removePermission[r,r':Role,p:Permission] {
18    r'.perms = r.perms - p
19 }
```

pred はPredicate、述語を表します。結果が真偽で表現されるのが述語の特徴です。機能を記述する際には、集合を返すものは **fun** で、真偽を返すものは **pred** を使って記述することになります。

上記の例では、rを事前条件、r'を事後条件としてのRoleとして考えます。addRoleでのu'.roles = u.roles + rは、「事後条件として、u'の持つrolesは、事前条件uの持つrolesにrを加えた集合」と読めます。

2.4.3 追加、削除機能の検証

ロールを追加して削除した場合、操作前と後では同じロール構成になっていなければなりません。これを表すには、以下の様に記述します。

```
/* ロールを追加して削除したのならば、操作前と後では同じロールを持つはず
   ただし、集合演算では、既に集合に入っている要素の追加も許されるため、
   no (u.roles) を追加する。
 */
assert AddAndRemoveRole {
   all disj u,u',u'':User,r:Role |
      (no (u.roles)) and addRole[u,u',r] and removeRole[u',u'',r]
      implies permissions[u] = permissions[u'']
}

check AddAndRemoveRole
```

assert は表明です。この表明は真である、と仮定されます。check コマンドによって反例を探索します。

- **all**： 「全ての」を表す量化子。| の前に指定した変数全てにおいて、| 以下の述語が成り立つことを表します。
- **disj**： u,u',u'' が同一アトムを示さないことを表します。Alloy Analyzer では、この指定を行なわなかった場合、u と u' が同じアトムの場合も含まれることになります。
- **implies**： 含意。「ならば」を表します。

上記の例は、

> 「別々の u,u',u'' および r の組み合わせ全てにおいて、
> u がロールを一つも持たず (**no** (u.roles))、
> u に r を追加したものが u' で (addRole[u,u',r])、
> u' から r を削除したものが u'' ならば (removeRole[u',u'',r])、
> u の権限の集合と u'' の権限の集合は等しい (permissions[u] = permissions[u''])」

ということを表します。
このようにして、仕様が正しいかどうかを確認していくことになります。
ここまでの完全な仕様は以下の通りです。

```
/*
   Alloy Analyzer 入門
   モデルを拡張する

   Role Based Access Control
   ロールに階層関係を追加

   author: Kenichi Kobayashi
```

```
 9  */
10
11  /* ユーザ。
12     0 以上のロールを持つ。
13  */
14  sig User {
15     roles: set Role
16  }
17
18  /* ロール。Admin,Guest など
19     0 以上の権限を持つ
20     0 以上の親のロールを持つ
21   */
22  sig Role {
23     perms: set Permission,
24     superRoles: set Role
25  }
26
27  /*割り当てられる操作。 Read,Write,ReadWrite など*/
28  sig Permission {}
29
30  /* 不変制約
31       ロールの階層構造がループしないこと。
32  */
33  fact {
34     no r:Role | r in r.^superRoles
35  }
36
37  /* ロールが階層化されているため、
38  スーパーロールのパーミッションも取得する */
39  fun permissions[u:User]: set Permission {
40     u.roles.*superRoles.perms
41  }
42
43  /* ユーザにロールを付与する。*/
44  pred addRole[u,u':User,r:Role] {
45     u'.roles = u.roles + r
46  }
47
```

```
48  /* ユーザからロールを剥奪する。*/
49  pred removeRole[u,u':User,r:Role] {
50    u'.roles = u.roles - r
51  }
52
53  /* ロールに権限を付与する。*/
54  pred addPermission[r,r':Role,p:Permission] {
55    r'.perms = r.perms + p
56  }
57
58  /* ロールから権限を剥奪する。*/
59  pred removePermission[r,r':Role,p:Permission] {
60    r'.perms = r.perms - p
61  }
62
63  /* ロールを追加して削除したのならば、操作前と後では同じロールを持つはず
64    ただし、集合演算では、既に集合に入っている要素の追加も許されるため、
65    (no (u.roles)) を追加する。
66   */
67  assert AddAndRemoveRole {
68    all disj u,u',u'':User,r:Role |
69      (no (u.roles)) and addRole[u,u',r] and removeRole[u',u'',r]
70      implies permissions[u] = permissions[u'']
71  }
72
73  check AddAndRemoveRole
```

3　最後に

3.1　本記事のまとめ

本稿では、Alloy Analyzer を用いたアジャイルモデリングを通して、形式手法によるモデリングの概観を解説しました。

成果物の精度を上げたい、品質を上げたい、と真剣にお考えの方に少しでも参考になれば幸いです。

筆者は現在、Formal Methods Forum というサークルを運営しています。ご興味がありましたら、ぜひご参加下さい。

3.2 参考文献

- Lightweight formal methods http://people.csail.mit.edu/dnj/publications/ieee96-roundtable.html
- Alloy Analyzer http://alloy.mit.edu/
- “Software Abstractions Logic, Language, and Analysis”, Daniel Jackson, MIT Press
- 「抽象によるソフトウェア設計」,Daniel Jackson 著, 中島 震 監訳, 今井 健男, 酒井 政裕, 遠藤 侑介, 片岡 欣夫 共訳, オーム社
- Formal Methods Forum http://groups.google.com/group/fm-forum
- ON-THE-FLY, LTL MODEL CHECKING with SPIN http://spinroot.com/spin/whatispin.html
- NuSMV:a new symbolic model checker http://nusmv.fbk.eu/