2003－SE－143　(8)
2003／7／17

セキュリティポリシー変更に関するデザイン解析

中島 震　　玉井 哲雄
法政大学 & さきがけ　　東京大学大学院

あらまし　セキュリティに関する考慮は、システム開発の初期段階から必要であるが、同時に、システム出荷ならびにサービスイン以降の要求事項変化に対応しなければならない。サービスイン以降の変更を容易にするために、設定ファイルに代表される宣言的な情報を修正することで要求変更に対応することが多い。しかし、現実のシステムでは、宣言的な情報だけでセキュリティ要求を実現することは難しい。アプリケーションプログラムに混在する手続き的な処理記述とあわさってはじめて所望の目的を達成することができる。セキュリティ要求の変更はプログラム変更を伴い、このような変更は当該アプリケーション機能と複雑に絡むため、適切に変更することが難しい。

本稿では、アクセス管理に関するセキュリティ要求の変更をデザインの段階で取り扱う。特に、具体的な基盤として JAAS を扱う。形式手法を用いた解析を行なうため、デザインを形式仕様言語 Alloy で表現する。宣言的な情報と手続き的な処理記述の双方を単一の枠組で表現・解析することで、互いの相互依存関係が明確になる。その結果、両方の観点を含む仕様記述に対する解析を行なうことが可能になる。特に、セキュリティ要求の変更に対するデザイン変更を議論する上で Alloy が有用であることを示す。

キーワード　セキュリティ要求、セキュリティポリシー、JAAS、コラボレーション、Alloy

Formal Design Analysis of Changes in Security Policy

Shin NAKAJIMA　　Tetsuo TAMAI
Hosei Univ. & PRESTO, JST　　The University of Tokyo

Abstract　Security is one of the major design issues that we should address in early stages of the system development. In addition, the system should have an extensible security. It is because the security requirements are often changed after the system is shipped and begins daily operations. In order to easily accommodate the system to the changes, the security enforcement mechanism is sometimes implemented in the application program level as seen in Java-based systems. Although the application-based approach has the advantages in view of the extensibility, portability, and performance, it sometimes makes the design complicated. A change in the security requirement may lead to a lot of changes spread over the design. In sum, the security is a “cross-cutting concern.”

We are interested in applying the formal specification and analysis techniques to the design regarding to the security requirements. In this paper, we argue that Alloy is shown useful for the analysis of changes in the security requirements. It owes much to the declarative specification style together with the constraint-based analysis mechanism. We present some technical details of a case using Alloy for the formal specification and analysis of JAAS.

Keywords　Security Requirements, Security Policy, JAAS, Collaboration, Alloy

1 はじめに

システム開発の初期段階からセキュリティに関する考慮が必要である。セキュリティに関しては、オープンなネットワークであるインターネットに接続する形態が増えていることから、ネットワーク経由での脅威に対抗するネットワークセキュリティが議論されることが多い。しかし、オペレーティングシステムの発展の歴史からもわかるようにマルチユーザのシステムではアクセス管理も重要な側面である。さらに、クライアントサーバ系のようにシステムが分散アーキテクチャの形をとるにしたがって、ネットワークとシステムの両方に跨るセキュリティ考慮が大切になってきている。

システム出荷以降にセキュリティに対する要求事項が変化することがある。ネットワークセキュリティでは、運用にしたがって新たな脅威 (セキュリティホール) が生じたり、新しいサービス提供に伴って、ファイアウォールの設定ポリシー変更が求められたりする。一方、システムセキュリティ、特に、アクセス管理についても、新しいサービス提供に加えて、利用者の変更、利用権限の変更、などが発生する。最近はシステムの可搬性や実行時性能の観点から、アクセス管理などのセキュリティ関連機能をアプリケーションプログラムのレベルで実現することが多くなっている [17]。

サービスイン以降の変更を容易にするために、設定ファイルなどに代表される宣言的な情報を修正することで要求変更に対応することが多い。しかし、現実のシステムでは、宣言的な情報だけでセキュリティ要求を実現することは難しい。アプリケーションプログラムに混在する手続き的な処理記述とあわさってはじめて所望の目的を達成することができる。セキュリティ要求の変更は結局のところプログラム変更を伴う。このような変更は当該アプリケーション機能と複雑に絡むため、適切に変更することが難しい。セキュリティ要求に関連するデザインを形式手法 [11] を適用して厳密に表現し、解析・検証することが望ましい。

本稿では、アクセス管理に関するセキュリティ要求の変更をデザインの段階で取り扱う。特に、具体的な基盤として JAAS [9] を扱う。形式手法を用いた解析を行なうため、デザインを形式仕様言語・解析ツール Alloy [4] で表現する。宣言的な情報と手続き的な処理記述の双方を単一の枠組で表現・解析することで、互いの相互依存関係が明確になる。その結果、両方の観点を含む仕様記述に対する解析を行なうことが可能になる。特に、セキュリティ要求の変更を考察する上で有用であることを示す。

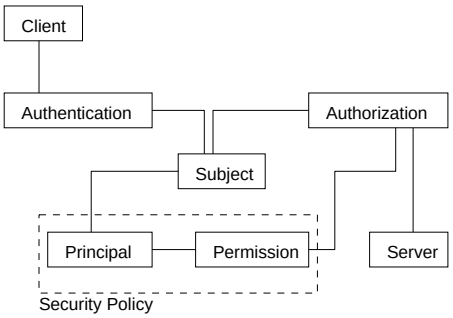

図 **1. Overview of JAAS Architecture**

2 セキュリティ要求に関するデザイン

JAAS (Java Authentication and Authorization Service) [9][13] を具体例として、セキュリティ要求の変更例を説明する。

2.1 JAAS

図 1 に JAAS の基本的な枠組を示した。本稿では JAAS アーキテクチャと呼ぶ。

JAAS は認証 (Authentication) と承認 (Authorization) の 2 つに分けることができる。認証サブシステムはシステム外部の利用者が当該システムを利用する権限を有しているか否かのチェックを行なう。利用者の認証を行なうためにログインのフレームワークを提供する。ログインを許可された利用者は システム内部では Subject として管理される。一般に一人の利用者、したがって一つの Subject は、システムを操作する複数の異なる権限を持つ。JAAS では、Principal を操作権限を有する実体とし、Subject が複数の Principal を持つとする。

承認サブシステムは、認証の結果の Subject を引き継ぐ。アクセス管理を行なうためにはサーバ資源の操作を許可する Permission 概念の導入が必要である。何らかの資源にアクセスするためには、Subject は適切な Permission を与えられた Principal を持たねばならない。認証プロセスで Sujbect がそのよう

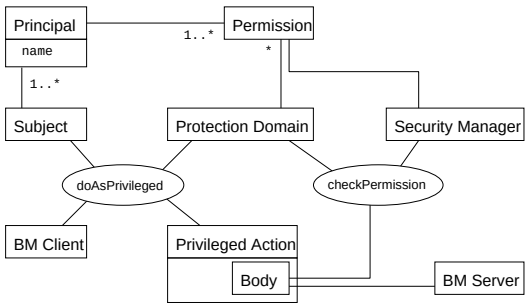

図 **2. Joint Actions of Authorization**

な Principal を持つ場合、当該の資源へのアクセスが許可される。なお、JAAS では、Principal とこれに操作許可を与えられた資源の関係の全体をセキュリティポリシーと呼び、この関係を宣言的な情報によってて表現し設定ファイルで保持管理する。簡単な例を示す。

```
grant principal BankingPrincipal ‘‘shin’’ {
  permission BankingPermission ‘‘setBalance’’
}
```

このような宣言的な情報をコンベンションにしたがったファイル名 (*.policy) のポリシーファイルに書き込む。この例は、名前が “shin” という BankingPrincipal に対して、“setBalance” という BankingPermission の操作許可を与えることを示す[1]。

JAAS による認証ならびに承認の機能振舞いを理解するためには、図 1 よりも詳しい説明が必要である。以降、具体的な議論を行なうために、承認サブシステムが行なうアクセス管理について説明する。

JAAS が提供する機能を整理し中心となる性質を表現するためにはデザイン記述の観点を決めることが大切である。本稿では、JAAS が複数の機能的なオブジェクトの集まりから構成される一種のオブジェクト指向フレームワークであるという点に着目し、関連するオブジェクトの集団的な振舞いあるいはコラボレーションの側面 [3][10] を明示することとした。より具体的には、JAAS 全体を互いに関連するジョイントアクションの集まりとして整理した。ジョイントアクションは Catalysis [16] が導入した考え方であり、複数オブジェクトが関与する機能単位を表す。図 2 にアクセス管理に関するダイアグラム表現を示した。

[1]クラス名は正しくは FQN でなければならない。

主要なジョイントアクションは、`doAsPrivileged` と `checkPermission` の 2 つである。BMClient がアクセス権限のチェックが必要な処理 (BMServer) を起動する場合、当該プログラムの利用主体である Subject がアクセス許可を持つことを確認しなければならない。

第 1 の `doAsPrivileged` ジョイントアクションは、アクセス権限チェックのための特別な特権コンテクストを生成することを示し、Subject、PrivilegedAction などが関係する。PrivilegedAction がラップする Body の中で実際に BMServer を起動する前に `checkPermission` による実行時のアクセス許可検査を挿入する。`checkPermission` ジョイントアクションは当該 Subject のアクセスが許可されるか否かを判定する仕組みとして ProtectionDomain を参照する。

アクセス許可判定はスタックインスペクションと呼ぶ機構を用いて実現する。Java アプリケーションプログラムではアクセス権限の判定を行なうために checkPermission メソッドを呼び出す。アクセスコントローラがメソッド実行を管理するスタックフレームの上から走査を開始し、各スタックエントリに対応する ProtectionDomain が要求された Permission を持つか否かを調べる。走査はスタックボトムに至るか、あるいは、doAsPrivileged メソッドによって特権化されたエントリに至るまで続け、すべてのエントリが Permission を持つ場合にチェックが成功する [2]。通常、アプリケーションプログラムでは、checkPermission に先立って、doAsPrivileged により、当該 Subject の Principal に与えられた Permission 情報を ProtectionDomain に関連つけておく。すなわち、これらの 2 つのメソッドが連携することで、当該 Subject がアクセス許可を得るようなプログラムを作成する。

2.2 セキュリティ要求の変更

セキュリティ要求とセキュリティポリシーという 2 つの用語の関係を整理しておく。一般にシステムが持つ要件あるいはデザインの中でセキュリティに関係するものを本稿ではセキュリティ要求と呼ぶ [3]。セキュリティポリシーという用語は JAAS での慣

[2]実際はもう少し複雑な仕様であり、doAsPrivileged がチェックする ProtectionDomain 群を明示的に指定する機能もある。

[3][15] によるセキュリティポリシーに関する要求。

習にしたがい設定ファイル (*.policy) が保持する情報のこととする。JAAS を用いるシステムでは、セキュリティ要求はセキュリティポリシー (設定ファイル) と JAAS フレームワークを用いた Java プログラムの両方で実現される。すなわち、セキュリティ要求は、宣言的な情報と手続き的な処理の組み合わせではじめて実現可能となる。

セキュリティ要求の変更は、サービスイン以降も発生することが考えられるため、宣言的な情報の書き換えだけで対応したい。実際、JAAS を用いる場合、設定ファイルとして表現可能なセキュリティポリシーを変更すれば良い。セキュリティポリシーは Principal とこれに操作許可を与えられた資源の関係の集まりである。既存の資源に新たな操作許可を持つ Principal を追加 (あるいは削除) する場合はセキュリティポリシーの書き換えで対応できる。

一方、新たな資源をシステムに組み込む場合、当該資源に対するアクセス処理を記述した箇所に適切なアクセス権限チェックの処理を埋め込む必要がある。新たに追加する処理は、局所性が必ずしも良いというわけではない。図 2 に示した 2 つのジョイントアクションに対応する Java プログラム断片をその起動順序を保って書き加える必要があり、複数箇所に変更が及ぶ。一種の cross-cutting concerns である。さらに、処理の流れを変更した場合のテストの際に、プログラムロジックの不具合であるか宣言的なセキュリティポリシーの設定ミスであるかの切り分けを含むため、双方の情報間の関係をきちんと把握しておく必要がある。JAAS をブラックボックスとして利用することは難しく、少なくとも図 2 に示した詳細レベルでの JAAS の基本的な仕組みに熟知する必要がある。

3 形式検証

セキュリティ要求変更は宣言的な情報と手続き的な処理記述といった異なる表現方法が絡む。本稿では、対象のセキュリティ要求変更に関係するデザイン変更の妥当性確認を行なうために形式手法 Alloy[4] を用いる方式を提案する。Alloy を用いた仕様表現と解析の具体例を説明する。

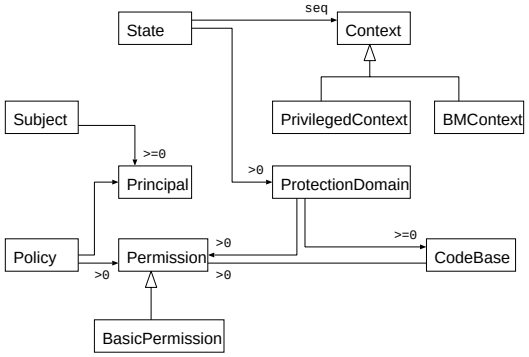

図 **3. Structural Relationship**

3.1 形式仕様記述

図 2 のジョイントアクションを Alloy で表現する。まず、参加オブジェクト間の静的な関係を整理する。次に、参加オブジェクトの状態変化として、ジョイントアクションが持つ機能を表現する。図 3 は参加オブジェクトの関係をダイアグラムで表現したものである。

第 1 に、基本的なデータオブジェクトを定義する。Subject、Principal、Permission といった基本的な用語を定義し、JAAS のセキュリティポリシーを表すために Principal と Permission の関係を保持する Policy を導入した。

```
sig Subject { principals: set Principal }
sig Principal { name: Name }
fact{ all p1,p2: Principal |
           p1.name = p2.name => p1 = p2 }
sig Name {}

sig Permission {}
static disj sig BasicPermission
               extends Permission {}

sig Policy {
  principals: set Principal,
  grant: principals ->+ Permission
}
```

Subject は Principal の集まりを保持すること、Principal は name で識別されること、を示す。また、Policy は、Principal と当該 Principal に付与された Permission の対応関係の集まりである。多重度を+ (one or more) にすることで、ひとつの Principal が複数の Permission を持つことを明示した。BasicPermission は static disj であり、他の Permission 項と区別されること (disj)、かつ一つに限定されること (static) から、実質的に “定数 ”として用いることができる。

第 2 に、JAAS のアクセスコントローラが行なうチェック処理の仕組みを表現するためにシステム状態 State を導入する。

```
sig State {
  stack: Seq[Context],
  map: Context ->? ProtectionDomain
}{ dom(map) = stack..SeqElems() }

sig ProtectionDomain {
  base: set CodeBase,
  map: base ->+ Permission
}

sig CodeBase {}
sig Context {}
static disj sig PrivilegedContext
               extends Context {}
```

State はスタックフレームを表現するために導入したもので Context の列からなる。各 Context は、Permission の集まりである ProtectionDomain と関連づけられている。上記では、JAAS の仕様を忠実に反映するために、CodeBase と Permission の対応関係の集まりとした。また、PrivilegedContext を導入して、doAsPrivileged 実行を表現するために必要となる特別な特権化された Context を表すこととした。

第 3 に、2 つのジョイントアクションに対応する機能を関数 (パラメータ化された論理式) として表現する。まず、doAsPrivileged については、セキュリティポリシーと実行主体が持つ情報が整合していること、ならびに、本アクションの実行によって、スタックに特権化されたエントリが作られることと解釈した。前者は、Policy と Subject の Principal が、重なりを持つこととして表現できる。後者は、State の状態変換 (state transformation) でスタック情報を更新する。

```
fun subjectToPerform
      (s1, s2: State, s: Subject,
            c: CodeBase, ps: set Policy)
{
  subjectAndPolicy(s,ps) &&
    expandSubject(s1,s2,ps,s,c)
}

fun subjectAndPolicy
       (s: Subject, ps: set Policy)
{
   some p: Policy |
      p in ps &&
      some (p.principal & s.principals)
}

fun expandSubject
       (s1: State, ps: set Policy,
         s: Subject, c: CodeBase): State
{
  sole d: ProtectionDomain |{
    d.base = c
    d.map = (c -> { y: Permission |
       all p: Policy, x: Principal |
         p in ps && x in s.principals &&
         y = p.grant[x] })
    result.map = s1.map + (PrivilegedContext -> d)
  }
  result.stack
     = SeqAddHead(s1.stack, PrivilegedContext)
}
```

2 番目の checkPermission ジョイントアクションはスタックインスペクションを行なう。スタックをトップから下向きに走査を開始し、走査の終了条件が満たされるまでに得たスタックエントリすべてについて当該パーミッションを保持するかをチェックする。

```
fun accessCheck
    (s1: State, u: Permission,
                c: CodeBase): State
{
  all i in collectConextIndices(s1.stack) |
    performCheckPermission(
      s1,(s1.stack)..SeqAt(i),u,c
    )
}

fun performCheckPermission
    (s: State, f: Context,
               u: Permission, c: CodeBase)
{
  let d = s.map[f] |{
     c in d.base &&  u in d.map[c]
   }
}
```

ここで、collectContextIndices は、Alloy による Seq の表現方法に引きずられたが、該当する Context の index の集合を求めるように表現した。

この時、2 つのジョイントアクションが正しい順序関係で実行するということは、中間状態 s2 (State 項) を経由して、上記の 2 つの関数の論理積で表現することができる。

```
fun correctBehavior
      (s1: State, s: Subject,
        c: CodeBase, ps: set Policy)
{
  some s2, s3: State | {
    subjectToPerform(s1,s2,s,c,ps)
    accessCheck(s2,s3,BasicPermission,c)
  }
}
```

関数 correctBehavior は、Subject の要素である Principal に与えられた Permission を Policy から集めて ProtectionDomain を構成する。その後、指定 Permission (この例では BasicPermission) が許可されるか

否かをスタックインスペクションの手法で確認する、ことを示す。スタックインスペクションを表現するためにはスタックが必要であり、この部分については Alloy 記述が少し複雑になった。しかし、全体の Alloy 記述規模は 300 行足らずである。これは、関連するドキュメント (たとえば、[9] や [13]) で説明に要する記述量と比べて、きわめてコンパクトであり、その結果、わかりやすいと考えることができる。

3.2 解析の例

前節で示した Alloy 記述を対象として種々の場合について解析する。

整合性の確認 第 1 に、作成した Alloy 記述が整合性を示すか、すなわち、何らかの解を持つかどうかは次のコマンドで確認することができる。

```
run correctBehavior for 3
                    but 1 Subject, 1 Policy
```

Alloy は correctBehavior を満たす項が存在するか探索する、すなわち、論理式のモデルを見つけようとする。一般的には、探索対象は無限集合になるためモデル探索が終了しない。そのため、モデル探索空間を有界化して、その範囲 (スコープ) 内で網羅的な探索を行なう。モデルが見つかれば当該の性質が整合性を持つことがわかる。一方、モデルが見つからない場合、スコープが小さいことが理由であるかもしれない。すなわち、もう少し大きなスコープで探索を行なえばモデルが見つかるかもしれない。したがって、モデルが見つからないことから、ただちに不整合であると結論できない。探索情報が不足していると考えるべきである。上記の例は、Subject と Policy が 1 である以外は 3 とするスコープを指定したことに相当し、この場合はモデルが見つかる。

テスト実行 第 2 に、Subject と Policy の両方に具体的なデータが設定されている場合の振舞いについて解析することを考える。テスト実行の 1 種である。これを行なうためには、Subject や Policy に属する “定数 ”を導入すれば良い。以下に Subject 定数の定義を一部示す。

```
static disj sig FirstPrincipal
                extends Principal {}
{ name = Ichi }

static disj sig Ichi extends Name {}

static disj sig Shin extends Subject {}
{ principals = FirstPrincipal }
```

意味的に整合しない Subject と Policy の定数に対して解析した場合、当然であるが、モデルが見つからない。反例を見つけるために、以下の式に対して check コマンドを実行する。

```
assert AccessViolation {
  all s2, s3: State, c: CodeBase |{
    subjectToPerform(
       NullState,s2,Shin,c,TestPolicy
    )
    accessCheck(s2,s3,BasicPermission,c)
   }
}

check AccessViolation for 3
                      but 1 Subject, 1 Policy
```

Alloy は当該の式を満たさない例、すなわち、反例を求める。これによって、モデルが見つからない場合の有界性から発生する情報の不足をカバーすることができる。

双方向の出力計算 最後に、Subject か Policy かのいずれか一方を定数として、前述の run コマンドによる解析を行なう。Policy を定数とする場合は、当該 Policy のもとで、アクセス権利を持つ Subject の条件を具体例として構成する。逆に、Subject を定数とした解析では、当該 Subject がアクセス権利を持つために必要な Policy の条件を求めることができる。

4 考察

前節の解析方法を組み合わせると、セキュリティ要求の変更に伴うデザインの解析が可能であることがわかる。以下、本稿で紹介した JAAS の基本ジョイントアクションに関する Alloy 記述が既にあるとする。

第 1 に、手続き的な処理の修正を伴うデザイン変更の場合、当該デザインのコラボレーションを JAAS の基本ジョイントアクションを含む組み合わせとして表現し、整合性の解析を行なえば良い。

第 2 に、セキュリティポリシー、すなわち、設定ファイルを変更する場合、テスト実行によって、

既に定義済みの利用主体がアクセス許可されるかどうかをチェックすることができる。不具合が見つかった場合、利用主体を変数化した出力計算の解析によって、セキュリティポリシーと整合した利用主体を求めることができる。セキュリティポリシーが満たすべき要求である場合には、利用主体を変更すればよい。これは、本稿では扱わなかった認証サブシステムで行なうべき処理に関係する。

最後に、定義済みの利用主体にアクセス許可を与え続けることが優先すべき要求であれば、セキュリティポリシーを変数化して出力計算する。設定ファイルに書き込むべきデータを得ることが可能となる。

以上、Alloy がモデル探索のために制約解消機構を用いているという特徴を活用することで、ひとつの仕様記述に対して、種々の観点からの解析を行なうことができた。また、Alloy は宣言的な仕様表現を採用していることから、セキュリティポリシーの表現に向いていることは容易にわかる。Alloy を用いて明示的な状態変換を記述することで手続き的な処理の流れも表現することができる。これによって、JAAS の本質的なデザイン側面を統一的に表すことができた。

5 関連研究

次に関連研究について簡単に述べる。複数オブジェクトの集団的な振舞いからなるコラボレーションの記述に形式手法を導入し解析・検証する試みがいくつか報告されている [11]。R.Helm たち [3] は、集団的な振舞いに着目した仕様言語を提案したが解析・検証は扱わなかった。Catalysis[16] はオブジェクトの集団的な振舞いを重要視する UML ベースの方法論であり、複数オブジェクトに跨る機能的な面を表現するために、ジョイントアクションを導入した。詳細な機能を表現するために OCL を用いる。中島と二木 [10] は、オブジェクトの集団的な振舞いを、代数仕様記述言語 CafeOBJ を用いて表現するものでラピッドプロトタイピングを可能とした。

Alloy[4] はデザインの構造的な側面の表現と解析に有効である。D. Jackson[5] はデザイン段階の集団的な振舞いを扱うものではなく、オブジェクトのエイリアス関係から生じる構造的な問題を扱っている。Alloy を用いて、エイリアス関係を表現し解析を行なっている。副作用を扱うために状態変換の考え方を導入するなど、そこで使われている技法はデザインの仕様表現にも有効である。A. Schaad と J.D. Moffett.[14] は、Alloy をセキュリティ関係の問題に適用した事例研究である。特に、ロールベースのアクセス管理方法 (RBAC) について、主として、構造的な側面の仕様表現と解析を行なった。

北山 [8] は Apache の設定ファイルの整合性を確認する問題に対して、形式仕様言語 Object-Z を使う方法を提案した。設定ファイルという従来、形式手法があまり扱わなかった問題に適用し有効性を示した。本稿では Java セキュリティポリシーという設定ファイルが保持する宣言的な情報を扱った。

Java のスタックインスペクションの問題は、D. Wallach たち [17] による整理からはじまって形式手法を用いた検証の研究がある。T. Jensen たち [6] は、LTL 式のモデル検査検証の問題に帰着させた。N. Nitta たち [12] は効率のよい検証アルゴリズムを示した。いずれも、JDK1.2 のセキュリティ機能 [2] を対象とするもので、JAAS までを含むものではない。

G. Karjoth[7] は JAAS の操作的な意味定義を遷移システムの枠組を使って形式的に表現した[4]。また、形式的な記述を用いて、RBAC などの高レベルのセキュリティポリシーを JAAS が表現可能であることを示した。しかし、本稿で扱ったような形式解析を行なっていない。本稿では、Alloy を用いて、構造的な側面からはじめてジョイントアクションとして整理した後、Alloy で定式化し解析した。特に、Alloy の制約ベースの解析方法を利用して、ひとつの仕様記述に対して、異なる有用な観点からの解析を行なうことができることを示した。

6 おわりに

本稿では、アクセス管理に関するセキュリティ要件の変更をデザイン段階で形式手法を用いて表現・解析する事例を報告した。具体的な基盤として JAAS を対象とし、デザインを形式仕様言語・解析ツール Alloy を用いた。その結果、宣言的な情報と手続き的な処理記述の双方を単一の枠組で表現・解析することで、互いの相互依存関係が明確にすることができ、かつ、両方の観点を含む仕様記述に対す

[4]関連研究を調査する段階で Karjoth の定式化を知った。本稿の Alloy 記述は全く独立に行なった形式化である。

る解析を行なうことが可能になった。また、Alloyが採用した仕様検証技法では、仕様を満たすモデル探索を行なうために制約解消機構を用いている。この特徴を生かすことで、ひとつの仕様記述に対して、種々の観点からの解析を行なうことができる。特に、セキュリティ要求の変更のデザインへの影響を考察する上で有用であることを述べた。

参考文献

[1] M.A. Bishop. *Computer Security: Art and Science*. 2003.

[2] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the JavaTM Development Kit 1.2. In *Proc. USENIX Symp. Internet Tech. Syst.*, December 1997.

[3] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proc. OOPSLA/ECOOP'90*, pages 169–180, 1990.

[4] D. Jackson, I. Shlyakhter, and M. Sridharan. A Micromodularity Mechanism. In *Proc. FSE-9*, September 2001.

[5] D. Jackson. Lightweight Analysis of Object Interactions. In *Proc. TACS 2001*.

[6] T. Jensen, D. Le Métayer, and T.Thorn. Verification of Control Flow Based Security Properties. In *Proc. IEEE Symp. S&P*, pages 89–103, 1999.

[7] G. Karjoth. An Operational Semantics of Java 2 Access Control. In *Proc. IEEE CSFW-13*, pages 224-232, July 2000.

[8] 北山 文彦. オブジェクトモデリングと Object-Z によるアプリケーション導入・配置時の構成・設定の検証. オブジェクト指向シンポジウム 2002, August 2002.

[9] C. Lai, L. Gong, L. Kovel, A. Nadalin, and R. Schemers. User Authentication and Authorization in the JavaTM Platform. In *Proc. 15th Computer Security Applications Conference*, December 1999.

[10] S. Nakajima and K. Futatsugi. An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ. In *Proc. ICSE'97*, pages 34–44, May 1997.

[11] 中島 震. オブジェクト指向デザインと形式手法. コンピュータソフトウェア, Vol. 18, No. 5, pages 17–46, September 2001.

[12] N. Nitta, Y. Takata, and H. Seki. An Efficient Security Verification Method for Programs with Stack Inspection. コンピュータソフトウェア, Vol. 19, No. 3, pages 20–38, May 2002.

[13] S. Oaks. *Java Security (2ed.)*. O'Reilly, 2001.

[14] A. Schaad and J.D. Moffett. A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. In *Proc. SACMAT'02*, pages 13–22, June 2002.

[15] F.B. Schneider. Enforceable Security Policies. ACM TISS, Vol.3, No.1, pages 30-50, February 2000.

[16] D. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML*. Addison Wesley, 1999.

[17] D. Wallach, D. Balfanz, D. Dean, and E. Felton. Extensible Security Architectures for Java. In *Proc. 16th Symp. Ope. Syst. Princ.*, pages 116–128, October 1997.