Pattern-based Approach to Modelling and Verifying System Security

Xiaoyu Zheng, Dongmei Liu
School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing, 210094, P.R. China
zxy961120@sina.com, dmliukz@njust.edu.cn

Hong Zhu, Ian Bayley
School of Engineering, Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, UK
hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

***Abstract*—Security is one of the most important problems in the engineering of online service-oriented systems. The current best practice in security design is a pattern-oriented approach. A large number of security design patterns have been identified, categorised and documented in the literature. The design of a security solution for a system starts with identification of security requirements and selection of appropriate security design patterns; these are then composed together. It is crucial to verify that the composition of security design patterns is valid in the sense that it preserves the features, semantics and soundness of the patterns and correct in the sense that the security requirements are met by the design. This paper proposes a methodology that employs the algebraic specification language SOFIA to specify security design patterns and their compositions. The specifications are then translated into the Alloy formalism and their validity and correctness are verified using the Alloy model checker. A tool that translates SOFIA into Alloy is presented. A case study with the method and the tool is also reported.**

***Keywords*-Security; Design patterns; Algebraic specifications; Formal verification; Model checking**

I. Introduction

The use of security patterns to solve system security design problems is the current best practice for security engineering [1], [2]. In this approach, well recognised solutions to various security concerns are documented as a collection of security patterns. A large number of security patterns have been recognised and documented in the past few decades; see e.g. [3], [4]. The overall security solution for a system can then be formed by composing security patterns that each address one of the security concerns [5]. However, errors may occur in selecting an inadequate solution to a specific security concern or in composing patterns together inappropriately, etc. A question that remains open is how to validate and verify that a given composition of security patterns is valid and really meets the overall security requirements of the system. To address the question, this paper proposes a formal approach based on algebraic specification.

The rest of this paper is organised as follows. Section II briefly reviews the related work. Section III outlines our proposed approach. Section IV is devoted to the formal modelling of security patterns and their compositions in the algebraic specification language SOFIA. Section V focuses on the automatic verification of security solutions specified in SOFIA using our prototype tool *A2A*. Section VI reports a case study with the security design of a blockchain-based crowd funding service application. Section VII concludes the paper with a discussion of future work.

II. Related Work

In this section, we briefly review related work in three areas: the formal verification of security solutions, software design patterns and algebraic specification techniques.

A. Verification of Security Patterns

Currently, security design patterns are commonly documented and specified in UML [6], [7]. UML is a semi-formal modelling language. It is widely used in software development and is the de facto standard for model-driven software engineering. Design patterns are easy to understand when described in UML, but due to its semi-formal nature, it is not precise enough for formal verification of the correctness of the patterns, nor their compositions and applications. Recently, formal methods have been employed to verify the correctness of a composition of security design patterns, for example, by employing model checking techniques [8]–[12].

To enable the formal verification of security patterns and their compositions and applications, two approaches have been advanced in the literature: extending the UML language definitions with formal semantics, and transforming UML diagrams into formal specifications. The formal models are then verified by employing model checking techniques. The following are among the most well-known in the literature:

- modelling security patterns through UMLSec [13] and using a model checker to check whether a security design specified in UMLSec meets the secrecy, integrity, authenticity and refinement conditions [14].
- transforming UML sequence diagrams to CCS as the specification of pattern composition and using model checker CWB-NC to check various properties specified by GCTL [8].
- transforming UML models of security patterns to Alloy [15] and using the Alloy Analyzer to check the incon-

sistencies and ambiguities within security patterns [10]. Five security patterns were studied.
- using High Level Petri-nets [16] and Colored Petri-nets [17] to model security patterns and to combine security patterns with existing models [11], [12].

The above works either focus on the individual security patterns or treat the composition of security patterns as a whole system. The information on how patterns are composed is omitted in the verification process. Consequently, although the consistency of the formally specified system can be verified, the validity of the use of a pattern in the system cannot be proved. Moreover, it is difficult to locate the faults in large and complex systems that consist of many patterns, if the verification process detects an error. Secondly, a UML model defines a design of the system without specifying the system's requirements. Therefore, the correctness of the design in terms of whether it meets the security requirements cannot be proved. Finally, model checking techniques verify the security properties of models, but not the correct implementation of models.

B. Software design patterns and pattern compositions

In a wider context, software design patterns have been an active research topic in recent decades. In addition to object-oriented design patterns [18], a wide range of software design patterns, including security design patterns, have been recognised, categorised and documented. Various approaches to improve the preciseness of pattern definitions have been proposed. These include formalisation of UML, development of modelling languages specifically for the definition of design patterns, and employment of formal logic and specification languages [19], [20]. However, as far as we know, there is no serious attempt to employ algebraic specifications in the same manner as this paper.

The composition of design patterns has also been studied by a number of authors. In [21], an algebra for pattern composition was proposed with a set of six operators on patterns for the composition and instantiation of design patterns. A complete set of algebraic laws for these operators were proved. They are useful for reasoning about pattern compositions and instantiations in pattern-oriented design of object-oriented systems as demonstrated in case studies [21].

The validity of pattern compositions was studied in [22], where the validity of a pattern composition was defined as its preservation of the features, semantics and soundness of the composed patterns. The conditions for a pattern composition to be valid were proved for various pattern operators. These lay a solid foundation for the work reported in this paper.

C. Algebraic specification

Algebraic specification is a formal method that supports specification refinement, verification and software testing. It has been widely used in the formal development of object-oriented software, software components and Web services [23]–[25]. Our empirical studies [26]–[28] demonstrated that algebraic specifications are easy to learn and to understand. They are suitable for the specification of large scale systems due to their modular structure. It enables patterns to be specified in a reusable and composable way. In particular, in [29], we proposed an approach to specify and verify service compositions in the so-called dual structure $\langle S_A, S_I \rangle$ of algebraic specifications, where S_A is the abstract specification that defines the requirements of composed services and S_I is the implementation specification that defines how the service is composed from other services. The *implementation* relationship that must be satisfied between S_A and S_I can be formally verified to ensure the correctness of the implementation of services through service composition. In this paper, we apply this dual structure to specify the compositions of security patterns.

Another attractive feature of algebraic specifications is that they can be used directly in automated software testing [23], including test case generation, test execution and test oracles. Thus, they can be employed to perform security testing to check if the implementation of a security solution is correct with respect to a formal model.

III. The Proposed Approach

This section outlines the proposed method for pattern-based security design. As shown in Figure 1, the process consists of two threads of tasks: modelling and verification.

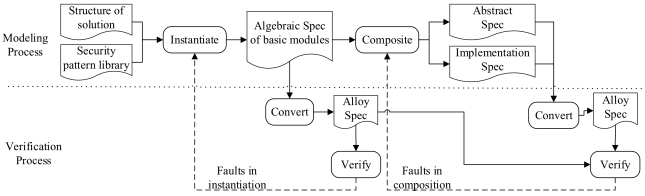


Figure 1. The Process

1) *Modeling* activities aim to construct a formal model of the security solution as an algebraic specification. It specifies the security solution in the dual structure of *abstract + implementation* specifications. The abstract specification defines the system's security requirements, while the implementation specification describes how security patterns are composed together. It also instantiates the security patterns to form specifications of basic components in the security solution; these are called modules in the sequel.

2) *Verification* activities transform the algebraic specifications into a set of correctness conditions to be verified. Two sets of correctness conditions are derived from the specifications: (a) *validity conditions* require that the composition of patterns preserves the features, semantics and soundness of the patterns used in the security solution; (b) *functional correctness conditions* require that the composition of the patterns as specified in the implementation specification

meets the security requirements as specified in the abstract specification.

IV. Specifying Security Patterns and Compositions

This section discusses how to specify security patterns and their compositions in the specification language SOFIA.

A. Specification of Security Patterns

A security pattern consists of a set of components, a set of operations on the components, a set of properties that the pattern satisfies (i.e. the security requirements it meets), and finally some usage information, such as the applicable platforms of the pattern. We use algebraic specification language SOFIA to specify security patterns.

In SOFIA [30], an algebraic specification of a computer system is a triple $\langle \mathbf{S}, \Sigma, Ax \rangle$, where $\mathbf{S} = \langle S, \rhd, \succ \rangle$, S is a set of sorts in the system representing the components in the system or abstract concepts used in the system, $\rhd$ and $\succ$ are the *extends* and *uses* relations on S. Σ and Ax are the set of signatures and the set of axioms of the specification, respectively. They define the syntax and semantics of the operators respectively. A system specification is decomposed into a number of modules such that each module specifies one sort, and contains a set of signatures and axioms associated to the sort.

Therefore, the specification of a pattern in SOFIA consists of a number of specification modules. Each component in the pattern is specified in a module with the component name as the sort. Each datatype used in the security pattern is also specified by a module in a similar way. A module with the name of the pattern specifies the components and datatypes used in the pattern, and a set of operations that the pattern provides. The signature defines the syntax of operations and the relationships between components. The axioms define the semantics of the operations and their dynamic properties as well as the security requirements. The usage information is only used for developers to identify the pattern and described in natural language.

For example, consider the access control pattern *Owner* shown in Figure 3. The components S in the pattern include (a) the collection of data entities owned by a particular user, (b) the database that stores all the data owned by different users. They are specified in module of sort *SOData* and *DataDB*. The datatype of the return value resulting when access to a piece of data is requested is also specified in a module, the sort *Return*. The operation set O includes (a) $owner\_verifier$ for verifying the ownership of a specific collection of data, and (b) $updatedata$ for updating data. Axiom 1 defines operation $owner\_verifier$ as follows: the operation returns *true* if the current user is the owner of data. The security requirement "only the owner of data can access the data" is stated as Axioms 2 and 3. After calling the data creation operation, the update operation is called to change the data that have just been created. If two different users perform the operations, the data cannot be updated; otherwise they can be updated successfully.

B. Specification of Pattern Compositions

In general, a composition of design patterns consists of a series of operations on the patterns composed; these operations include instantiations of patterns, restriction and extensions of patterns in the context of the application, superposition of patterns together, lifting or flattening patterns on the variables of the pattern, etc. [21]. Due to the modular structure of SOFIA specifications, these operators can be realised simply by manipulating the signatures and axioms of the specification modules. For the sake of space, this paper will not detail how exactly to adapt the operations for SOFIA, instead we will illustrate the process with an example.

For example, Figure 2 shows the security solution for the access control system of a blockchain-based application. It consists of four basic modules in Layer 0 and two composition modules in Layer 1 and Layer 2, respectively. Edges with arrows depict the dependency relation between modules, while edges with diamonds depict the composition relation between modules.

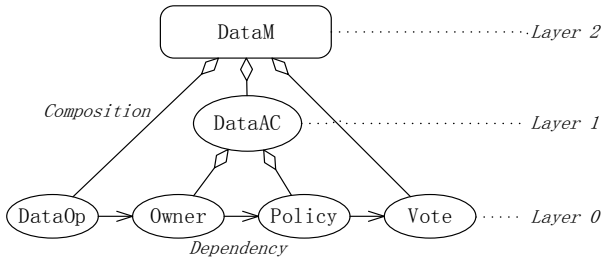


Figure 2. The Structure of an Access Control System

The specifications of components and abstract data types used in a security pattern are first instantiated by substituting the sorts with those of the concrete data types actually used in the application. Additional attributes may be added into the specification to link to the context of the application.

For example, consider the *Owner* pattern used in a blockchain application to manage medical data. The abstract data type *SOData* is instantiated to a sort called *MedicalCase*, where the attribute $owner$ in *SOData* is systematically replaced by $patient\_id$. It is also extended by adding new attributes $visit\_time$ and $hospital$ to the module.

```
Spec SOData;  //before instantiating and extension
uses Integer;
 Attr
  owner: Integer;
End

Spec MedicalCase;  //after instantiating and extension
uses Integer, Date, String;
 Attr
  patient_id:Integer;
```

```
  visit_time:Date;
  hospital:String;
End
```

Replacing one attribute with another causes it to be systematically replaced in the axioms as well. For example,

```
//before instantiation:
o.owner_verifier(db,uid,i)=true, if d.owner=uid;

//after instantiation:
o.owner_verifier(db,uid,i)=true, if d.patient_id=uid;
```

The algebraic specification of a composition module consists of the abstract specification $\langle \boldsymbol{S}^A, \Sigma^A, Ax^A \rangle$ and the implementation specification $\langle \boldsymbol{S}^I, \Sigma^I, Ax^I \rangle$.

The abstract specification specifies the semantics and behaviour of an operation with a set of axioms that only contain the operations defined in its signature Σ^A.

The implementation specification, in contrast, defines how the operations of the abstract specifications invoke operations defined in the components and datatypes. For example, the following axioms of *DataM* specify the function and implementation of the operation *updatedata_policy*, respectively.

```
//abstract
dac.updatedata_policy(...).content = s,
 if dac.access_verifier(...) = true;

//implementation
dac.updatedata_policy(...) = dac.dop.updatedata(...),
 if dac.access_verifier(...) = true;
```

The axiom in the abstract specification states that the content of the data after the operation is *s* if the current user passes the permission check. The axiom in the implementation specification states that the permission verification operation *access_verifier* in *DataAC* is called first, then the update operation *updatedata* in *DataOp* is called if *access_verifier* returns *true*.

V. Derivation of Verification Obligations

To verify that a pattern-based security solution is correct, we identify two types of verification obligations:

- *Validity of the pattern composition*: For a pattern composition to be valid, it must preserve the features of the patterns composed together, and preserve the semantics and soundness of the patterns [22]. In the context of the algebraic specification of security patterns, this means that the axioms of the modules in the specification of the patterns are consistent with each other after they are instantiated and extended.
- *Satisfaction of security requirements*: A composition of a set of components is correct if and only if the implementation specification of the composition satisfies all the properties specified in the abstract specification of the system [29]. This means that the axioms in the implementation specification plus the axioms in the component specifications after instantiation and extension entail the axioms in the abstract specification.

This section presents the transformation of security design specifications in SOFIA into verification obligations in the formalism that can be automatically checked by the model checking tool Alloy.

A. Transformation of Basic Module

Let $\langle \boldsymbol{S}, \Sigma, Ax \rangle$ be the SOFIA specification of a basic module, where $\boldsymbol{S} = \langle S, \rhd, \succ \rangle$, $\Sigma = \{\Sigma^s \mid s \in S\}$, and $Ax = \{Ax^s \mid s \in S\}$. It is translated into an Alloy specification that consists of the following elements:

1) $\mathscr{S} = \langle \mathcal{S}, > \rangle$: the classes $\mathcal{S}$ in the module and the extends relationships $>$ between them.
2) $\mathscr{M} = \{\mathscr{M}^s | s \in \mathcal{S}\}$: the data members in the module, where $\mathscr{M}^s$ is the set of data members in the class s.
3) $\mathscr{E}$: the set of enumeration data in the module.
4) $\mathscr{F} = \{\mathscr{F}^s | s \in \mathcal{S}\}$: the constraints on the data members, where $\mathscr{F}^s$ is the constraints on the data members in $\mathscr{M}^s$. For each $f \in \mathscr{F}^s$, $f = \langle V_f, p_f \rangle$, where V_f is the set of variables and p_f is a predicate on those variables.
5) $\mathscr{O}$: the set of operations in the module. Each $op \in \mathscr{O}$ has the form $op = \langle \varphi, In, Out \rangle$, where φ is operation name, In and Out are the sets of input and output parameters.
6) $\mathscr{P} = \{\mathscr{P}^{op} | op \in \mathscr{O}\}$: the assertions on the operations in the module, where $\mathscr{P}^{op}$ is the set of assertions on operation op.
7) $\mathscr{A}$: the behaviour properties of the module. For each $a \in \mathscr{A}$, $a = \langle V_a, p_a \rangle$, where V_a is the set of variables, $p_a = \langle seq, e_a \rangle$ is a predicate that asserts the equation e_a holds after executing the sequence seq of operations on the system.

The following are the rules to derive Alloy specifications from SOFIA specifications of basic modules.

Rule 1: $\mathscr{S}$ in the Alloy specification contains the set $\mathcal{S}$ of classes that is the same as the sort set S in SOFIA specification, and the extends relation $>$ the same as $\rhd$ in SOFIA. There is no explicit definition of the uses relations in Alloy so the relation $\succ$ in SOFIA specifications is omitted in the translation.

Rule 2: For the signature part of each sort $\Sigma^s \in \Sigma$, constants operators (*Const*) are added to the enumeration set $\mathscr{E}$, attribute operators (*Attr*) are translated into the data members $\mathscr{M}^s \in \mathscr{M}$ of signature s, and other operators (*Retr&Tran*) are added to operation set $\mathscr{O}$.

Rule 3: For each axiom $ax_{attr} = \langle V, e \rangle$ defining constraints on an attribute $attr$, ax_{attr} is translated into a fact $f = \langle V_f, p_f \rangle$ and added to $\mathscr{F}$, where $V_f = V$ and p_f is derived from the conditional equation e.

Rule 4: For each axiom $ax_{op} = \langle V, e \rangle$ defining the properties of an operation, the conditional equation e is translated into a predicate $p^o \in P^o$, where the variables in p^o are the input and output parameters of operation o.

Rule 5: For each axiom $ax_{seq} = \langle V, e\rangle$ defining the properties of a sequence of operations, ax_{seq} is translated into an assertion $a \in A$, where the variable set V is translated into aV and conditional equation e is translated into $seq \wedge ae$.

These rules are implemented by Algorithm 1. Table I lists the operations used in the algorithms.

Table I
SYMBOLS USED IN ALGORITHMS

Symbol	Description
$ax.v$	The set of variables V in axiom ax
$ax.e$	The conditional equation e in axiom ax
$\prod_{op}(ax)$	The operation in axiom ax
$\prod_{oplist}(ax)$	The set of operations in axiom ax
$\prod_{out}(op)$	The set of output parameters of operation op

Figure 3(a) is an algebraic specification in SOFIA and (b) is the result of its translation into Alloy specification.

B. Transformation of Composition Module

A composite module in SOFIA is translated into an Alloy specification that consists of the following elements:

1) $\mathscr{O}$: the set of operations in the module.
2) $\mathscr{P} = \{\mathscr{P}^o | o \in \mathscr{O}\}$: the set of predicates that defines the implementation of operations in the module, where $\mathscr{P}^o$ is the predicate defining the implementation of operation o.
3) $\mathscr{A}$: the set of assertions on the behaviours of the operations. It defines the security requirements of the module. $\mathscr{A} = \mathscr{A}_{op} \cup \mathscr{A}_{seq}$, where $\mathscr{A}_{op} = \{\mathscr{A}^o_{op} | o \in \mathscr{O}\}$. $\mathscr{A}^o_{op}$ is the set of assertions defining the external function of operation o. A_{seq} is the set of assertions defining the security requirements.

The following are the rules that extract operations, predicates and assertions from SOFIA specifications of pattern compositions, and translate them into Alloy.

Rule 1: The signatures Σ^A of the abstract specification are added to the operation set $\mathscr{O}$.

Rule 2: For each axiom $ax^A_{op} = \langle V, e\rangle$ that defines the properties of an operation op in the abstract specification, it is translated into an assertion $a = \langle V_a, e_a\rangle$ and added to $\mathscr{A}$, where the variable set V is translated into V_a and the conditional equation e is translated into e_a.

Rule 3: For each axiom $ax^A_{seq} = \langle V, e\rangle$ that defines the properties of a sequence seq of operations in an abstract specification, it is translated into an assertion $a = \langle V_a, e_a\rangle$ and added into $\mathscr{A}$, where the variable set V is translated into V_a, the conditional equation e is translated into e' and $e_a = \langle seq, e'\rangle$.

Rule 4: The signatures Σ^I of the implementation specification are added to the operation set $\mathscr{O}$.

Rule 5: For each axiom $ax^I = \langle V, e\rangle$ in the implementation specification, the conditional equation e of the axiom is translated into a predicate p^o and added into $\mathscr{P}^o$, where the variables in p^o are the input and output parameters of operation o.

The above rules are implemented by Algorithm 2.

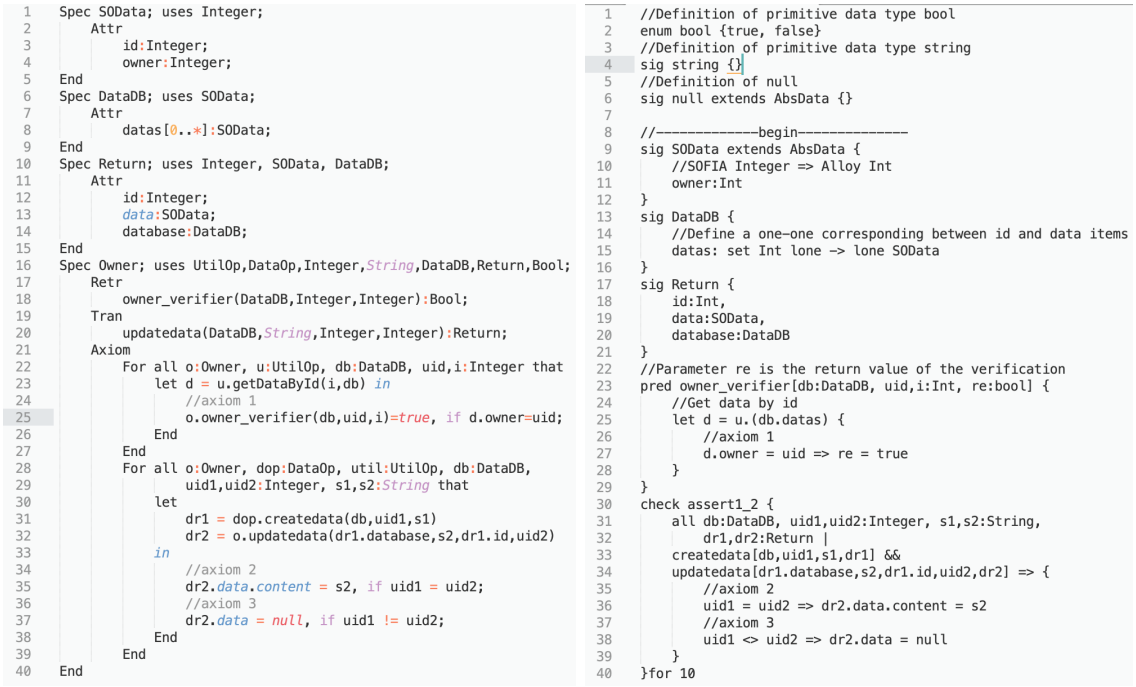
Algorithm 1 Conversion of Basic Module

Input: The SOFIA specification $\langle \mathbf{S}, \Sigma, Ax\rangle$
Output: The Alloy specification $\langle \mathscr{S}, \mathscr{M}, \mathscr{E}, \mathscr{F}, \mathscr{O}, \mathscr{P}, \mathscr{A}\rangle$
//Step 1: Convert each sort and its signatures into Alloy signature, enumeration or operation according to Rule 1 and 2.
for each $s \in S$ **do**
$\mathcal{S} \leftarrow \mathcal{S} + s$; $R_> \leftarrow R_> + \triangleright^s$; $\mathscr{E} \leftarrow \mathscr{E} + \Sigma^s_{Const}$;
$\mathscr{M}^s \leftarrow \Sigma^s_{Attr}$; $\mathscr{M} \leftarrow \mathscr{M} + \mathscr{M}^s$;
$\mathscr{O} \leftarrow \mathscr{O} + \Sigma^s_{Retr} + \Sigma^s_{Tran}$;
end for
$\mathscr{S} \leftarrow \langle \mathcal{S}, R_>\rangle$;
for each $ax \in Ax$ **do**
//Step 2: Convert each axiom in Ax_{attr} into Alloy fact according to Rule 3.
if $ax \in Ax_{attr}$ **then**
$V \leftarrow ax.v$; $p \leftarrow ax.e$; $\mathscr{F} \leftarrow \mathscr{F} + \langle V, p\rangle$;
end if
//Step 3: Convert each axiom in Ax_{op} into an equation in Alloy predicate according to Rule 4.
if $ax \in Ax_{op}$ **then**
$op \leftarrow \prod_{op}(ax)$; $\mathscr{P}^{op} \leftarrow \mathscr{P}^{op} + ax.e$;
end if
//Step 4: Convert each axiom in Ax_{seq} into an Alloy assertion according to Rule 5.
if $ax \in Ax_{seq}$ **then**
$Seq = \prod_{oplist}(ax)$; $V \leftarrow ax.v$; $e \leftarrow ax.e$;
for each $op \in Seq$ **do**
$V \leftarrow V \cup \prod_{out}(op)$;
end for
$\mathscr{A} \leftarrow \mathscr{A} + \langle V, \langle Seq, e\rangle\rangle$;
end if
end for
//Step 5: Add the Alloy predicates consisting of a collection of equations to the predicate set $\mathscr{P}$.
for each $op \in Seq$ **do**
$\mathscr{P} \leftarrow \mathscr{P} + \mathscr{P}^{op}$;
end for
return $\langle \mathscr{S}, \mathscr{M}, \mathscr{E}, \mathscr{F}, \mathscr{O}, \mathscr{P}, \mathscr{A}\rangle$;

C. Verification of Pattern Composition

The verification of a security solution consists of two stages. The first stage verifies that each basic module is consistent. The second stage checks the composition modules for their validity and correctness.

In Stage 1, for each basic module in the specification, each operation defined by the predicate is executed first to check whether the operation's pre/post conditions contradict other axioms.



```
Spec SOData; uses Integer;
    Attr
        id:Integer;
        owner:Integer;
End
Spec DataDB; uses SOData;
    Attr
        datas[0..*]:SOData;
End
Spec Return; uses Integer, SOData, DataDB;
    Attr
        id:Integer;
        data:SOData;
        database:DataDB;
End
Spec Owner; uses UtilOp,DataOp,Integer,String,DataDB,Return,Bool;
    Retr
        owner_verifier(DataDB,Integer,Integer):Bool;
    Tran
        updatedata(DataDB,String,Integer,Integer):Return;
    Axiom
        For all o:Owner, u:UtilOp, db:DataDB, uid,i:Integer that
            let d = u.getDataById(i,db) in
                //axiom 1
                o.owner_verifier(db,uid,i)=true, if d.owner=uid;
            End
        End
        For all o:Owner, dop:DataOp, util:UtilOp, db:DataDB,
                uid1,uid2:Integer, s1,s2:String that
            let
                dr1 = dop.createdata(db,uid1,s1)
                dr2 = o.updatedata(dr1.database,s2,dr1.id,uid2)
            in
                //axiom 2
                dr2.data.content = s2, if uid1 = uid2;
                //axiom 3
                dr2.data = null, if uid1 != uid2;
            End
        End
End
```

(a) Specification in SOFIA

```
//Definition of primitive data type bool
enum bool {true, false}
//Definition of primitive data type string
sig string {}
//Definition of null
sig null extends AbsData {}

//-------------begin--------------
sig SOData extends AbsData {
    //SOFIA Integer => Alloy Int
    owner:Int
}
sig DataDB {
    //Define a one-one corresponding between id and data items
    datas: set Int lone -> lone SOData
}
sig Return {
    id:Int,
    data:SOData,
    database:DataDB
}
//Parameter re is the return value of the verification
pred owner_verifier[db:DataDB, uid,i:Int, re:bool] {
    //Get data by id
    let d = u.(db.datas) {
        //axiom 1
        d.owner = uid => re = true
    }
}
check assert1_2 {
    all db:DataDB, uid1,uid2:Integer, s1,s2:String,
        dr1,dr2:Return |
    createdata[db,uid1,s1,dr1] &&
    updatedata[dr1.database,s2,dr1.id,uid2,dr2] => {
        //axiom 2
        uid1 = uid2 => dr2.data.content = s2
        //axiom 3
        uid1 <> uid2 => dr2.data = null
    }
}for 10
```

(b) Alloy Specification Generated

Figure 3. Example of Translation

Then each assertion is rewritten into a predicate by converting the universally quantified variables of the assertion into the variables of the predicate; see the example below.

```
//assert before rewriting
assert assertion {
  all v1:type1, v2:type2 | op[v1,v2] => v1=v2
}

//predicate after rewriting
pred predicate [v1:type1, v2:type2] {
  op[v1,v2] => v1=v2
}
```

The predicate is checked to ensure that the operation sequence and the security requirements defined in the assertion are consistent.

Finally each assertion is verified to ensure that the module meets the security requirements defined in the assertion.

In Stage 2, the composition modules are verified. This consists of the following three steps.

Step 1: Verifying the preservation of soundness.

Assume that each pattern is sound, i.e. they can all be satisfied. The soundness of the composition of the patterns requires that the specification is satisfiable, too. Therefore, for each operation *op*, we use Alloy to execute *op* described by the predicate P^o to generate an instance that satisfies the predicate. If the instance does not exist, this indicates that the specification of operation *op* is not satisfiable. Thus, the specification does not preserve soundness, and the specification is faulty.

Step 2: Verifying the preservation of features.

The features of a pattern are specified as axioms in the specification of the patterns and their components. Verification of the feature preservation condition of validity is done by proving that these axioms are also true in the composition specification. Therefore, we verify each assertion that defines the behaviours and properties of the operations. First, the assertion is rewritten into a predicate in the same way as for the verification of basic modules. The resulting predicate after rewriting has the same variables, operations, and constraints as the original assertion. It is executed in Alloy to generate an instance. If an instance can be generated from the predicate, it means that the predicate is consistent with all other axioms. Then the assertion is verified by generating counterexamples within the given scope. If there is no counterexample, the assertion is true within the given scope. In that case, the predicate P^o defining the implementation of operation *op* satisfies the axioms in the abstract specification that defines the external function of operation *op* as defined by the assertion, thereby proving that the composition preserves the features.

Step 3: Verifying the functional correctness.

The verification of functional correctness of pattern composition in the context of security design involves verifying each assertion in A_{seq} that defines the security requirements. The verification process is similar to Step 2. First, the assertion is rewritten into a predicate to generate instances. If an instance can be generated, the assertion does not contradict the axioms. Then, we attempt to generate counterexamples

Algorithm 2 Conversion of Composition Module

Input: The SOFIA specification consisting of $\boldsymbol{S}_I$ and $\boldsymbol{S}_A$
Output: The Alloy specification $\langle \mathscr{O}, \mathscr{P}, \mathscr{A} \rangle$

//Step 1: Add the signatures of $\boldsymbol{S}_A$ and $\boldsymbol{S}_I$ to the operation set $\mathscr{O}$.
$\mathscr{O} \leftarrow \Sigma^A \cup \Sigma^I$;
//Step 2: Convert each axiom in set Ax^A into an Alloy assertion according to Rule 2 and 3.
for each $ax \in Ax^A$ **do**
  $seq \leftarrow \prod_{oplist}(ax)$; $V \leftarrow ax.v$; $e \leftarrow ax.e$;
  for each $o \in seq$ **do**
    $V \leftarrow V \cup \prod_{out}(o)$;
  end for
  //If the axiom defines an operation o, add it to $\mathscr{A}^o_{op}$; otherwise, add it to $\mathscr{A}_{seq}$
  if $ax \in Ax^A_{op}$ **then**
    $o \leftarrow seq\,[0]$; $\mathscr{A}^o_{op} \leftarrow \mathscr{A}^o_{op} + \langle V, \langle seq, e \rangle \rangle$;
  else $\mathscr{A}_{seq} \leftarrow \mathscr{A}_{seq} + \langle V, \langle seq, e \rangle \rangle$;
  end if
end for
//Step 3: Convert each axiom in Ax^I into an equation in Alloy predicate according to Rule 5.
for each $ax \in Ax^I$ **do**
  $o = \prod_{op}(ax)$; $p^o \leftarrow p^o + ax.e$;
end for
//Step 4: Add each predicate and assertion on operation o to the predicate set $\mathscr{P}$ and assertion set $\mathscr{A}_{op}$.
for each $o \in \mathscr{O}$ **do**
  $\mathscr{P} \leftarrow \mathscr{P} + p^o$; $\mathscr{A}_{op} \leftarrow \mathscr{A}_{op} + \mathscr{A}^o_{op}$;
end for
$\mathscr{A} \leftarrow \mathscr{A}_{op} + \mathscr{A}_{seq}$;
return $\langle \mathscr{O}, \mathscr{P}, \mathscr{A} \rangle$;

to the assertion. If there is no counterexample, the assertion is true within the given scope and the specification must meet the security requirements defined in the assertion.

There are several common errors in security design, which include violating the operation's pre/post conditions and errors in the invocations of operations. These faults can be detected by checking the predicates and assertions with Alloy. Our case study shows that the above verification process can detect most faults in a security solution; see Section VI.

D. Prototype Tool A2A

This subsection presents a prototype tool called *A2A*. It has been developed in Java to transform specifications in SOFIA into Alloy formalism. As shown in Figure 4, A2A contains three main components.

1) *SOFIA Parser* parses the specification written in SOFIA, checks that it is syntactically well-formed, and checks that the equations in the axioms are type correct.

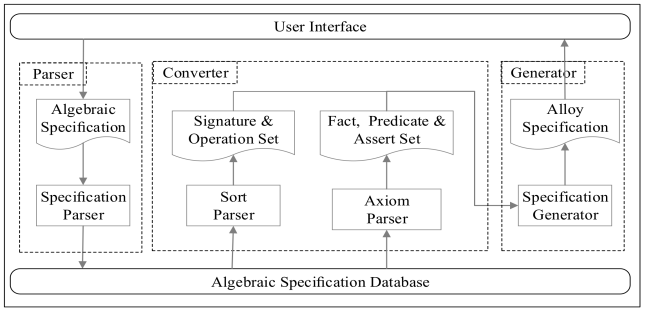


Figure 4. The Structure of Tool A2A

2) *Specification Converter* takes the parse tree of a SOFIA specification as input, and transforms that specification into an Alloy specification according to Algorithms 1 and 2. The Sort Converter analyses the signature of the algebraic specification and converts it into sorts and operations in the Alloy formalism. The Axiom Converter analyses the axioms, extracts the operations and adjusts the structure of the equations, and finally, converts the axioms into facts, predicates or assertions in Alloy formalism.

3) *Alloy Generator* generates an Alloy specification document as the end result of the conversion.

Figure 5 shows A2A's graphic user interface. The buttons enable the users to load SOFIA specifications, convert them to Alloy code, and save the result into a file. The specification panel, on the left hand side, displays the SOFIA specification to be converted. The result panel, on the right, displays the resulting Alloy code. The message panel at the bottom of the window shows the status of transformation and the error messages, if any.

VI. Case Study

This section presents a case study of the proposed technique and the tool A2A with the security design of a real world service-oriented system.

The case study aims to answer two research questions:

1) *Usability*: Is the proposed approach practical? In particular, how complex are the specifications of security patterns, their compositions and the abstract specifications of security requirements? How time consuming is the verification of a security solution against an abstract specification?
2) *Effectiveness*: Is the proposed approach effective to ensure the correctness of security designs? In particular, can it detect faults in the specification of the security requirements and the specification of the design of security solutions represented as composition of security design patterns.

This section reports the findings from the case study to answer these questions.

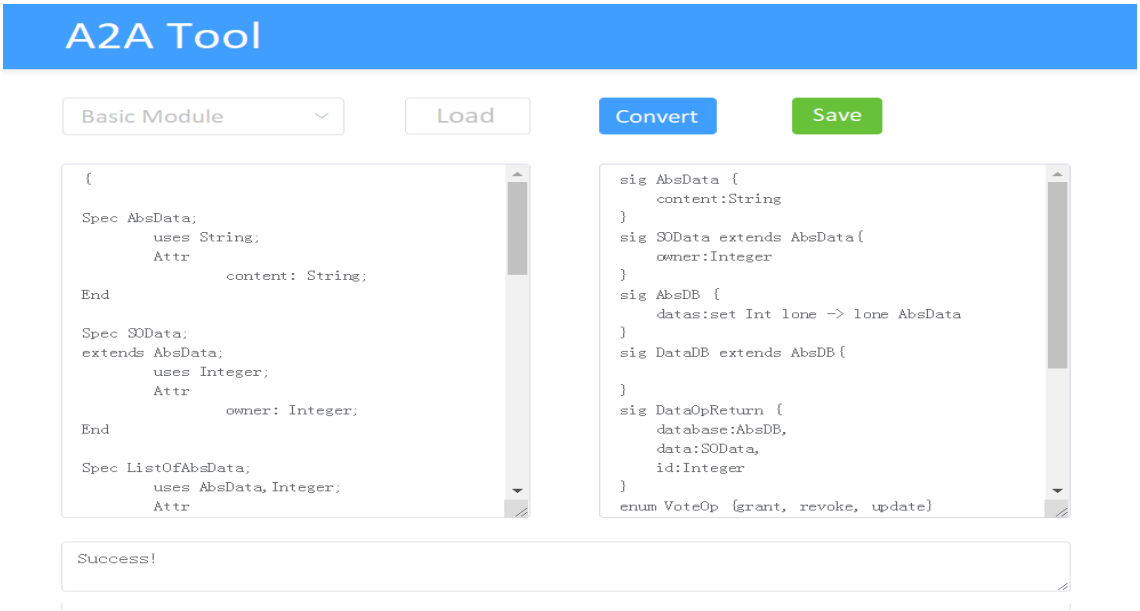


Figure 5. A2A's Graphic User Interface

A. The Subject

The subject of the case study is the Blockchain-enabled CrowdFunding application *BCF* developed by IBM [31]. The functions of *BCF* include viewing crowdfunding events, donating to events and viewing donation records. The case study is to make a security design that enhances the security of the application by adding several new security mechanisms. The enhanced security solution is modelled and formally specified using the algebraic specification language SOFIA. The specification is then converted into the Alloy formalism to verify the correctness of the security design. The extended system is called *EBCF* in the sequel.

There are three subsytems in *EBCF*:

- *event management* is in charge of the creation and termination of fund raising events;
- *account management* provides functions of recharge and withdrawal of fund to accounts;
- *donation management* makes donations and viewing donation records.

The structure of *EBCF* is depicted in Figure 6.

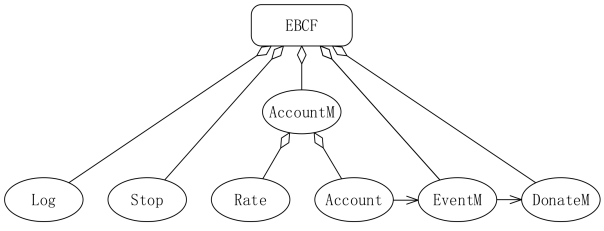


Figure 6. The Structure of EBCF

B. Specification of BCF in SOFIA

Four security design patterns are used in the design of the security solution:

- *Log* for logging the operations on the user's account,
- *Emergency Stop* for disabling sensitive operations,
- *Rate Limit* for regulating how often an operation can be called consecutively, and
- *Owner* for checking the ownership of events/entities.

These patterns and all the modules in *EBCF* have also been specified in SOFIA[1]. The specification consists of a number of packages such that each may contain several specification modules. Figure 7 shows the specification packages and their relationships. Basic packages $Basic$ and $UtilOp$ specify the basic data types and utility operations of the system. On top of them are level 1 packages, which include (a) Log which is an instantiation of security pattern Log, (b) $Stop$ an instantiation of pattern Emergency Stop, (c) $Rate$ an instantiation of pattern $RateLimit$, (d) $Account$, for managing the user's account, (e) $EventM$ an instantiation of pattern $Owner$ for managing events, and (f) $DonateM$, for making donations to events. On top of the level 1 are packages of the abstract specification $AccountM$ and its implementation specification $AccountMImp$ of the composition module $AccountM$ for limiting how frequently the withdraw function can be called. Finally, the top-level packages $EBCF$ and $EBCFImp$ specify the whole system *EBCF*.

[1]The specification in SOFIA and the generated code in Alloy can be found at https://github.com/SP-Case/SP-Case-Study

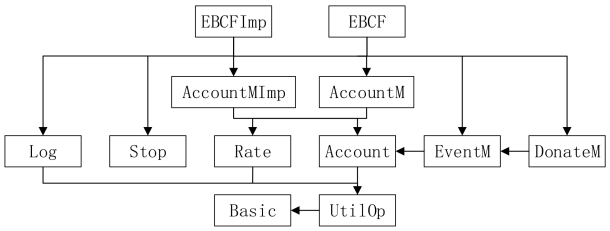


Figure 7. The Specification Structure of EBCF

C. Complexity of The Specifications

Our case study found that the specifications of security design patterns, their compositions and abstract security requirements are very simple and easy to write. Table II gives the number of sorts, operations, and axioms in various SOFIA specification units and the total number of lines of each specification unit.

Table II
STATISTICS OF SOFIA SPECIFICATIONS

Spec Unit	#Sort	#Ops	#Axioms	LOC
Log	5	2	10	64
Stop	2	2	6	34
Rate	7	3	14	99
Account	4	3	20	91
EventM	5	5	44	151
DonateM	5	1	19	80
AccountMImp	2	1	6	31
AccountM	1	1	11	41
EBCFImp	5	4	25	97
EBCF	1	4	48	151

Our case study also found that in most cases the Alloy code generated from a specification unit in SOFIA is of similar sizes to the SOFIA specification. Table III gives the number of modules and asserts and the total number of lines of the Alloy module generated by A2A.

Table III
STATISTICS OF ALLOY SPECIFICATIONS

Module	#Sig	#Fact	#Pred	#Assert	LOC
Log	4	0	2	5	46
Stop	1	0	2	2	27
Rate	6	2	3	3	81
Account	3	1	3	5	97
EventM	4	2	5	11	153
DonateM	4	1	1	3	70
AccountM	1	0	1	11	58
EBCF	4	0	4	48	244

However, as shown in Figure 3, specifications in SOFIA are more readable and easier to write than Alloy code. In particular, Alloy code is essentially execution instructions for the Alloy model checker. In contrast, SOFIA is more abstract and independent of the verification mechanism.

Another advantage of writing specification in SOFIA over writing in Alloy's formalism directly is that when the composition is in a hierarchical structure as we proposed in this paper, the axioms in an abstract specification of composition are the target to be verified from the axioms of its design specification and the axioms of its components. Once these axioms are verified, they can be used to verify the composition at a higher level. If the specifications are written in Alloy formalism directly, this means the axioms must be written twice; once as the verification; and once as facts or assertions to verify other properties. Moreover, in addition to verifying the functional correctness of the design solutions as discussed above, each of the axioms is also used more than once for different proof obligations: the soundness of the composition of components and patterns together, the soundness of instantiation of a pattern.

The proposed approach enables verifications be performed on a smaller scale instead of putting all the modules together. Our case study shows that time needed to execute Alloy to verify the security design of EBCF is feasible.

We used the model checker Alloy Analyzer to verify the predicates and assertions in basic modules Log, Stop, Rate, Account, EventM and DonateM. Then the predicates and assertions that specify the operations in composition modules AccountM and EBCF are checked to verify the validity and correctness of pattern compositions, while the assertions specifying the security requirements are checked to verify the security properties of the system. For example, the specification of EBCF in the Alloy formalism consists of 4 predicates, which are 48 assertions specifying the properties of operations and system's behaviour as the security requirements. During the verification, instances were successfully generated for both the predicates defining operations and for the rewritten assertions. No counterexample was generated for the assertions. This means that the composition of patterns meets the security requirements defined by the specifications.

Table IV shows the time taken to verify various assertions for a number of instances ranging from 10 to 20. The time that Alloy model checker takes to verify an assertion is exponential in the size of the module. Our approach helps to reduce the number of modules and thus the number of assertions in each verification. The data show that the approach is practically useful.

D. Effectiveness to Detect Errors

In order to study the effectiveness of error detection, we injected some faults into the axioms of SOFIA specifications and verified these faulty specifications to evaluate the effectiveness of the proposed method in detecting faults.

The faults were injected into algebraic specifications according to the specification mutation operators defined by Woodward [32]. These mutation operators are designed to simulate common errors made by software engineers. Typical examples of the operators include replacing constants, variables and operations in axioms, adding or deleting

Table IV
EXECUTIONS TIMES TO VERIFY AN ASSERT (MS)

Module	Assert	10 Ins	15 Ins	20 Ins
Stop	assert1_2	37	49	138
Account	assert1	61	277	615
Account	assert2_3	61	342	178
AccountM	withdraw1_3	93	105	108
EBCF	withdrawWithLog1_5	178	1970	1060
EBCF	stopAndPayWithLog1_7	191	199	710
EBCF	donateWithLog1_6	220	883	1474
Rate	assert1_2	339	512	997
EBCF	rechargeWithLog1_4	385	584	410
EBCF	assert23_26	456	966	413
Account	assert4_5	554	4676	6850
AccountM	assert1_8	637	1679	2245
DonateM	assert1_3	765	1930	1859
Log	assert1_5	847	1332	3887
EBCF	assert1_6	3494	24714	12175
EBCF	assert7_13	6207	13494	15546
EventM	assert1_4	8648	102794	383472
EventM	assert5_7	26194	131138	340009
EventM	assert8_11	81292	421072	1051382
EBCF	assert14_22	85418	1134374	3461973
Total	(ms)	216077	1843090	5285501
	(minutes)	3.60	30.72	88.09

axioms, and deleting the if conditionals of axioms. Most of these faults change the operation pre/post conditions or their invocations, making the specification fail to meet the security requirements. The following is such an example.

```
//axiom 1
es.estop(uid,s).value = false, if s.admin = uid;
//Fault injected axiom 1 (replace constant)
es.estop(uid,s).value = true, if s.admin = uid;

//axiom 2
am.withdraw(...).ure = a.subBalance(...);
//Fault injected axiom 2 (replace operation)
am.withdraw(...).ure = a.addBalance(...);
```

Axiom 1 specifies the operation *estop* in the Stop module and requires that the attribute *value* of sort *Status* is set to *false* if the user who invokes the operation is admin. In axiom 1 with the injected fault, the constant *false* is replaced with *true*, resulting in an error in the post condition definition of *estop*. Axiom 2 specifies the operation *withdraw* in the AccountM module and states that the operation *withdraw* will invoke the *subBalance* operation on Account to decrease the user's balance. In faulty axiom 2, *subBalance* is replaced with *addBalance*, resulting in invocation of the wrong operation in *withdraw*.

A total of 150 faults were injected into the specification and each generated a mutant specification. The data show that more than 90% of faults can be detected; see Table V.

However, 12 out of 150 mutant specifications go undetected. In these 12 mutants, 5 mutants are equivalent to the original specification. These are modifications of the specification that do not change the behaviour of the operation; these include replacing a constant that has no effect on the operation result, adding a redundant axiom or an axiom with an if condition that can never be satisfied.

For example, consider the axioms of operation *subBalance* in Account. They assert that the account's balance is decreased by the value of input param *n* and the new balance is returned, if the value of *n* is less than or equal to the current balance; otherwise the balance is not changed and the operation returns $null$. The mutation operator inserts an axiom that if the operation does not return $null$, the balance returned by the operation is greater than or equal to 0. This inserted axiom is redundant and has no effect on the behaviour of the *subBalance* operation.

Table V
VERIFICATION RESULTS OF DEFAULT INJECTION

Fault \ Result	Basic module	Composition module	Total	Rate(%)
Replace constant	11/12	11/11	22/23	95.7
Replace variable	20/23	18/18	38/41	92.7
Replace operation	6/6	4/4	10/10	100
Delete axiom	22/25	11/11	33/36	91.2
Add axiom	6/8	8/10	14/18	77.8
Delete condition	11/12	10/10	21/22	95.5
Total	**76/86**	**62/64**	**138/150**	**92**

```
let  u = util.getDataById(uid,db) in
  a.subBalance(uid,n,db).user.balance = u.balance-n,
  if u.balance >= n;
  a.subBalance(uid,n,db).user = null,
  if u.balance < n;
  //injected axiom
  a.subBalance(uid,n,db).user.balance >= 0,
  if a.subBalance(uid,n,db).user <> null;
```

Another finding of the case study is that the faults in certain security scenarios cannot be detected if there is no axiom in the specification about such scenarios. In other words, the incompleteness of a formal specification cannot be detected. The remaining 7 undetected faults all belong to this type. For example, the mutant is formed by deleting the condition of the axiom for event management for paying to the event creator after the termination of the event. This condition prevents repeat payment by checking whether the event has been successfully terminated before. Since there are no axioms that define the scenario of repeated termination, this fault cannot be detected.

VII. CONCLUSION AND FUTURE WORK

In this paper, we developed a method for the formal specification and verification of security patterns and their compositions for online service systems using the algebraic specification language SOFIA. A tool A2A has been implemented to transform SOFIA specification into Alloy formalism so that the verification of security designs can be automated. A case study with a crowdfunding application demonstrated the effectiveness of the proposed approach.

We are currently constructing a library of specifications of security patterns. We are also conducting more case studies with real-world examples of service-oriented applications.

Existing works on the formal verification of design pattern compositions have all been on the consistency of the

resultant system or the validity of the uses of patterns in the system. As far as we know, there is no work either on proving functional correctness of pattern compositions in general, nor on functional correctness of security designs. Our approach has the advantage of algebraic specifications in which structural, functional and behavioural properties can be specified in a unified framework. We believe that our approach can be extended for more general design patterns. This is an interesting direction for future work.

After proving that an instantiation and composition of patterns satisfy the functional requirements of the system as we did in this paper, we can reduce the proof obligations that an implementation of a system satisfies the overall security requirements that each module in the system satisfies the axioms in its corresponding specification module. This will significantly reduce the difficulty and complexity of formal verifications. Another possibility for future work is to investigate how to realise this.

Another advantage of algebraic specification is that it enables automated testing of software against formal specifications [23], [24]. These test automation techniques can be applied to the implementation of system security design formally specified in algebraic specifications. A possibility for future work is to conduct experiments on this automated testing.

ACKNOWLEDGEMENT

The work is partially supported by the National Science Foundation of China (Grant No. 61502233).

REFERENCES

[1] C. Blackwell and H. Zhu, Eds., *CyberPatterns: Unifying Design Patterns with Security Patterns and Attack Patterns*. Springer, 2014.

[2] B. Hamid and D. Weber, "Engineering secure systems: Models, patterns and empirical validation," *Computers & Security* 77, pp. 315–348, 2018.

[3] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," in *Proc. of PLoP'97*, vol. 2, 1997.

[4] P. H. Nguyen, K. Yskout, T. Heyman, J. Klein, R. Scandariato, and Y. Le Traon, "Sospa: A system of security design patterns for systematically engineering secure systems," in *Proc. of MODELS 2015*. IEEE, 2015, pp. 246–255.

[5] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *Proc. of IWBOSE 2018*, pp. 2–8.

[6] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[7] E. Fernandez-Buglioni, *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[8] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Information and Software Technology* 52(3), pp. 274–295, 2010.

[9] B. Hamid, S. Gürgens, and A. Fuchs, "Security patterns modeling and formalization for pattern-based development of secure software systems," *Innovations in Systems and Software Engineering* 12(2), pp. 109–140, 2016.

[10] A. K. Dwivedi and S. K. Rath, "Formalization of web security patterns," *INFOCOMP* 14(1), pp. 14–25, 2015.

[11] X. He and Y. Fu, "Modeling and analyzing security patterns using high level petri nets." in *Proc. of SEKE 2016*, pp. 623–627.

[12] A. Norta, R. Matulevcius, and B. Leiding, "Safeguarding a formalized blockchain-enabled identity-authentication protocol by applying security risk-oriented patterns," *Computers & Security*, 2019.

[13] J. Jürjens, "UMLSec: Extending UML for secure systems development," in *Proc. of UML'02*, Springer, pp.412–425.

[14] H. Schmidt and J. Jürjens, "Connecting security requirements analysis and secure design using patterns and umlsec," in *Proc. of AISE 2011*, Springer, pp. 367–382.

[15] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT Press, 2012.

[16] X. He, "A comprehensive survey of petri net modeling in software engineering," *Int'l Journal of Soft. Eng. and Knowl. Eng.* 23(5), pp. 589–625, 2013.

[17] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured petri nets and cpn tools for modelling and validation of concurrent systems," *Int'l J. on Software Tools for Technology Transfer* 9(3-4), pp. 213–254, 2007.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] R. B. France, D.-K. Kim, S. Ghosh, and E. Song, "A UML-based pattern specification technique," *IEEE Trans. Softw. Eng.* 30(3), pp. 193–206, 2004.

[20] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *J. of Systems and Software* 83(2), pp. 209–221, Feb. 2010.

[21] H. Zhu and I. Bayley, "An algebra of design patterns," *ACM Trans. on Soft. Eng. and Meth.*, 22(3), Article 23, 2013.

[22] ——, "On the composibility of design patterns," *IEEE Trans. Soft. Eng.* 41(11), pp. 1138–1152, Nov. 2015.

[23] D. Liu, Y. Liu, X. Zhang, H. Zhu, and I. Bayley, "Automated testing of web services based on algebraic specifications," in *Proc. of SOSE 2015*, IEEE, pp. 143–152.

[24] D. Liu, X. Wu, X. Zhang, H. Zhu, and I. Bayley, "Monic testing of web services based on algebraic specifications," in *Proc. of SOSE 2016*, IEEE, pp. 24–33.

[25] X. Zhang, D. Liu, H. Zhu, Y. Chen, B. Lan, and Y. Sun, "A test execution engine for automated web services testing based on algebraic specifications," *Computer Engineering & Science* 46(1), pp. 114–121, 2018.

[26] H. Zhu and B. Yu, "An experiment with algebraic specifications of software components," in *Proc. of QSIC 2010*. IEEE, pp. 190–199.

[27] D. Liu, H. Zhu, and I. Bayley, "Applying algebraic specification to cloud computing–a case study of infrastructure-as-a-service goGrid," in *Proc. of ICSEA 2012*, pp. 407–414.

[28] ——, "A case study on algebraic specification of cloud computing," in *Proc. of 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 269–273.

[29] Y. Chen, D. Liu, H. Zhu, B. Lan, and J. He, "Algebraic specifications of service composition," *Computer Engineering & Science* 2018(6), p. 18, 2018.

[30] D. Liu, H. Zhu, and I. Bayley, "Sofia: An algebraic specification language for developing services," in *Proc. of SOSE 2014*, IEEE, pp. 70–75.

[31] IBM, "blockchain-enabled-crowdfunding," https://github.com/IBM/blockchain-enabled-crowdfunding, last access: Dec 14, 2019.

[32] M. R. Woodward, "Errors in algebraic specifications and an experimental mutation testing tool," *Software Engineering Journal* 8(4), pp. 211–224, 1993.