Towards A General Theory of Patterns

Hong Zhu
Department of Computing and Communication Technologies
Oxford Brookes University, Oxford, OX33 1HX, UK
Email: hzhu@brookes.ac.uk

***Abstract* -- As knowledge of solutions to recurring design problems, a large number of software design patterns (DP) has been identified, catalogued and formalized in the past decades. Tools have been developed to support the application and recognition of patterns. However, although the notions of pattern in different subject domains carry a great deal of similarity, we are in lack of a general theory that applies to all types of design patterns. This paper is based on our previous work on formalization of OO DPs and an algebra of pattern compositions. We propose a generalization of the approach so that it can be applied to other types of DPs. In particular, a pattern is defined as a set of points in a design space that satisfy certain conditions. Each condition specifies a property of the instances of the pattern in a certain view of the design space. The patterns can then be composed and instantiated through applications of operators defined on patterns. The paper demonstrates the feasibility of the proposed approach by examples of patterns of enterprise security architecture.**

Keywords – design space; design pattern; enterprise security systems; formal methods.

I. INTRODUCTION

Since 1980s, much work has been reported in the literature on the patterns of OO software designs. Here, a design pattern (DP) is a piece of codified knowledge of design solutions to recurring design problems. A pattern-oriented design methodology has been advanced by the identification and catalogue of patterns, the formalization of them and the development of techniques and tools for formal reasoning about patterns and automating pattern oriented design and code recovery. Its success in improving OO design has also fostered research on patterns of other aspects of software design, such as interface, architecture and fault tolerant designs. The notion of patterns has also been extended to other phases of software lifecycle, such as analysis patterns in requirements analysis, architectural patterns in software architectural design, process patterns in software process modelling, test patterns in software testing, etc.

In a more general context, the notion of pattern has been investigated in many subject areas of computer science. In particular, security patterns [1, 2] and attack patterns have been identified and catalogued in the study of computer security. However, although the notions of patterns in different subject areas carry a great deal of similarity, we are in lack of a general theory that applies to all types of patterns.

In this paper, we propose an approach to generalize our previous work on the formalization of OO DPs and algebra of pattern compositions and instantiations. We will also explore the applicability of the general theory to security and identify the new problems in the study of security patterns.

II. RELATED WORKS

A. OO Design Patterns

In the past decade, several formalisms for formally specifying OO DPs have been advanced [3]. In spite of differences in these formalisms, the basic underlying ideas are quite similar. That is, patterns are specified by constraints on what are its valid instances via defining their structural features and sometimes their behavioural features too. The structural constraints are typically assertions that certain types of components exist and have a certain configuration of the structure. The behavioural constraints, on the other hand, detail the temporal order of messages exchanged between the components.

Therefore, in general, a DP P can be defined abstractly as a tuple $<V, Pr_S, Pr_D>$, where $V=\{v_1:T_1, .. , v_n:T_n\}$ declares the components in the pattern, while Pr_S *and* Pr_D are predicates that specify the structural and behavioural features of the pattern, respectively. Here, v_i's in V are variables that range over the type T_i of software elements, such as class, method, and attribute. The predicates are constructed from primitive predicates either manually defined, or systematically induced from the mate-model of software design models [4]. The semantics of a specification is the ground formula $\exists V_1. (Pr_S \wedge Pr_D)$.

The notion of *pattern conformation,* i.e, a concrete design *D conforms* to a pattern *P*, or *D is an instance of P*, can be formally defined as logic entailment $D \models \exists V \cdot Pr$ (i.e. the statement $\exists V \cdot Pr$ is true on D), where $Pr = Pr_S \wedge Pr_D$ and we write $D \models P$. Consequently, for patterns P_i, i=1,2, $\exists V_1 \cdot Pr_1 \Rightarrow \exists V_2 \cdot Pr_2$ means pattern P_1 is a specialization of pattern P_2 and we have that for all designs D, $D \models P_1$ implies that $D \models P_1$. In other words, reasoning about the specialization relation between patterns and the conformation of designs to patterns can be performed in formal logics.

In [5], we have proposed the following operators on DPs for pattern composition and instantiation.

- *Restriction* $P[C]$: to impose an additional constraint C to pattern P;
- *Superposition* $P_1 * P_2$: to require the design to conform to both pattern P_1 and P_2;
- *Generalisation* $P \Uparrow x$: to allow an element x in pattern P become a set of elements of the same type of x.
- *Flatten* $P \Downarrow x$: to enforce a set x of element in the pattern P to be a singleton.
- *Lift* $P \uparrow x$: to duplicate the number of instances of pattern P in such a way that the set of components in each copy satisfies the relationship as in P and the copies are configured in the way that element x serves as the primary key as in a rela-

tional database.

- *Extension* $P\#(V \bullet C)$: to add components in V into P and connect them to the existing components of P as specified by predicate C.

Using these operators, pattern oriented design decisions can be formally represented [6]. A complete set of algebraic laws that these operators obey has also been established so that the result of design decisions can be worked out formally and automatically. Moreover, with the algebraic laws, the equivalence between different pattern expressions can be proven formally and automatically through a normalization process. For example, we can prove that the equation $P[\|X\| = 1] = P \Downarrow X$ holds for all patterns P.

B. Design Space

Generally speaking, a design space for a particular subject area is a space in which design decisions can be made. Each concrete design in the domain is a point in this space. Understanding the structure of a design space of a particular domain plays a significant role in software design [7]. Three approaches to represent design spaces have been advanced in software engineering research:

- *Multi-dimensional discrete Cartesian space*, where each dimension represents a design decision and its values are the choices of the decision.
- *Hierarchical structure*: where nodes in a tree represent a design decision and alternative values of the decision are the branches, which could also be dependent design sub-decisions [8].
- *Instance list*: where a number of representative instances are listed with their design decisions.

In the General Design Theory (GDT) proposed by Yoshikawa [9, 10], a design space is divided into two views: one for the observable (structural) features of the artefacts, and the other for functional properties. These two views are linked together by the instances in the domain. These instances show how combinations of structural properties are associated to the combinations of functional properties. These two views are regarded as topological spaces and the links as continuous mappings between them. By doing so, two types of design problems can be solved automatically.

- *Synthesis problem* is to find a set of the structural features as a solution that has certain functional features that are given as design requirements.
- *Analysis problem* is to find out the functional properties from an object's structural properties.

The existing work on OO DPs can be understood in the GDT very well, which also provides a theoretical foundation for the approach proposed in this paper. However, existing approaches to the representation of design spaces cannot deal with the complexity of software design satisfactorily. Thus, we propose to use meta-modelling.

C. Meta-Modelling

Meta-modelling is to define a set of models that have certain structural and/or behavioural features by means of modelling. It is the approach that OMG defines UML and model-driven architecture [11]. A meta-model can be in a graphic notation such as UML's class diagram, or in text format, such as GEBNF, which stands for graphic extension of BNF [12].

In GEBNF approach, meta-modelling is performed by defining the abstract syntax of a modelling language in BNF-like meta-notation and formally specifying the constraints on models in a formal logic language induced from the syntax definition. Formal reasoning about meta-models can be supported by automatic or interactive inference engines. Transformation of models can be specified as mappings and relations between GEBNF syntax definitions together with translations between the predicate logic formulas.

In GEBNF, the abstract syntax of a modelling language is a 4-tuple $<R, N, T, S>$, where N is a finite set of non-terminal symbols, and T is a finite set of terminal symbols. Each terminal symbol, such as String, represents a set of atomic elements that may occur in a model. $R \in N$ is the root symbol and S is a finite set of syntax rules. Each syntax rule can be in one of the following two forms.

$$Y ::= X_1 \mid X_2 \mid \cdots \mid X_n \quad (1)$$

$$Y ::= f_1 : E_1, f_2 : E_2, \cdots, f_n : E_n \quad (2)$$

where $Y \in N$, $X_i \in T \cup N$, f_i*'s* are field names, and E_i*'s* are syntax expressions, which are inductively defined as follows.

- C is a basic syntax expression, if C is a literal instance of a terminal symbol, such as a string.
- X is a basic syntax expression, if $X \in T \cup N$.
- $\underline{X@Z.f}$ is a basic syntax expression, if X, $Z \in N$, and f is a field name in the definition of Z, and X is the type of f field in Z's definition. The non-terminal symbol X is called a referential occurrence.
- E^*, $E+$ and $[E]$ are syntax expressions, if E is a basic syntax expression.

The meaning of the above meta-notation is informally explained in Table 1.

TABLE 1 MEANINGS OF GEBNF NOTATION

Notation	Meaning
X^*	A set of elements of type X.
$X+$	A non-empty set of elements of type X.
$[X]$	An optional element of type X.
$\underline{X@Z.f}$	A reference to an existing element of type X in field f of an element of type Z.

Informally, each terminal and non-terminal symbol denotes a type of elements that may occur in a model. Each terminal symbol denotes a set of predefined basic elements. For example, the terminal symbol String denotes the set of strings of characters. Non-terminal symbols denote the constructs of the modelling language. The elements of the root symbol are the models of the language.

If a non-terminal symbol Y is defined in the form (1), it means that an element of type Y can be an element of type X_i, where $1 \le i \le n$.

If a non-terminal symbol Y is defined in the form (2), then, Y denotes a type of elements that each consists of n elements of type $X_1,\ldots, X_n$, respectively. The k'th element in the tuple can be accessed through the field name f_k, which is a function symbol of type $Y \rightarrow X_k$. That is, if a is an element of type Y,

we write $a.f_k$ for the k'th element of a.

Given a well-defined GEBNF syntax $G = <R, N, T, S>$ of a modelling language L, we write $Fun(G)$ to denote the set of function symbols derived from the syntax rules. From $Fun(G)$, a predicate logic language can be defined as usual (C.f. [13]) using variables, relations and operators on sets, relations and operators on types denoted by terminal and non-terminal symbols, equality and logic connectives *or* $\vee$, *and* $\wedge$, *not* $\neg$, *implication* $\rightarrow$ and *equivalent* $\equiv$, and quantifiers *for all* $\forall$ and *exists* $\exists$.

III. The Proposed Approach

A. Overview

The proposed approach consists of the following aspects.

- *Definition of design space*.

We will use GEBNF-like meta-notation to define a meta-model as the design space. The meta-model will defines a number of views. In each view, the meta-model will define a number of types of component elements in the subject domain and relations and properties of the elements.

From a GEBNF-like meta-model, a predicate logic language will be induced as in [12]. In this language, the sorts of the elements are the types defined in the meta-model. The primitive relation symbols, function symbols and predicate symbols are the functions, relations and properties defined for the design space.

- *Specification of patterns in a design space.*

The patterns in a design space can then be specified formally using the induced predicate logic in the same way as we define OO DPs. That is, each pattern is defined by a predicate in the induced predicate logic language.

Patterns can also be defined as compositions and instantiations of existing patterns by applying the operators on patterns defined in [5]. We believe that the algebraic laws proved in [6] should also hold for such design spaces. Therefore, the proofs of properties of patterns can be performed in the same way as in OO design patterns.

B. Definition of Design Spaces

We represent a design space in the following form.

```
DESIGN SPACE <Name>;
    <Element type definitions>;
    <View definitions>
END <Name>
```

An element type definition is in the form of GEBNF formula (1). For example, the following is the definition of elements in an object oriented design.

```
DESIGN SPACE OODesign;
  TYPE
    Class ::= name: String, attrs: Property*, ops: Operation*;
    Property  ::= name: String, type:Type;
    Operation ::= name: String, params: Parameter*;
    Parameter ::= name: String, type: Type;
  VIEW …
END OODesign.
```

A view defines a set of properties of the element types and relationships between them together with some constraints. For example, the following is the structural view of OO designs at class level. The constraint states that inheritance is not allowed to be in cycles.

```
VIEW Structure;
BEGIN
  PROPERTY
    Features:
      {Class | Operation | Property} ->
        {Abstract, Leaf, Public, Private, Static, Query, New}*;
    Direction:
      Parameter -> {In, Out, InOot, Return};
  RELATION
    association, inherits, composite, aggregate: Class x Class;
 CONSTRANT
    FOR ALL c, d : Class THAT
      NOT (inherits(c,d) AND inherits(d,c)).
END Structure;
```

A view may also contain additional element types. For example, the behavioural view of OO design contains new types of elements such as *messages*, *lifelines*, and *execution occurrences*, *frameworks*.

```
VIEW Behaviour;
  TYPE
    Message::= OpId:string, params: ActuralParameter*;
    Life-line::=  ObjName:string, ClassName:string, Finish: [INT];
    ExecutionOcc::= lifeline: Life-line, start, finish: INT;  …
  PROPERYTY
      Type: Message ->{synchronous, asynchronous, return};
  RELATION
    Message: Lifeline x Lifeline;
    ActiveExec: ExecutionOcc x Lifeline;
    Trigs: Message x ExecutionOcc;
  END Behaviour;
```

C. Specification of Patterns

A pattern can be defined in two ways. The first is to define a pattern as a set of points in a design space in the following form.

```
PATTERN <Name> OF <Design space name>;
  COMPONENT {<Var>: <TypeExp>}+
  CONSTRANT  {IN <View name> VIEW: <Predicate> }*
END <Name>
```

For example, the *Composite* pattern in the Gang-of-Four catalogue can be defined as follows.

```
PATTERN Composite OF OODesign;
  COMPONENT
    leaves: SET OF Class;
    component, composite: Class;
  CONSTRAINT
    IN Structure VIEW
      inherits(composite, component);
      composite(component, composite);
      FOR ALL c IN leaves THAT inherits(c, components);
      component.features = {abstract};
    IN Behaviour VIEW …
END Composite.
```

The second way is to define a pattern as a composition or instance of other patterns by applying the pattern composition operators to existing ones. For example, the following defines a generalised version of the *Composite* pattern.

```
PATTERN G-Composite OF OODesign;
  COMPONENT
    components: SET OF Class;
  EQUALS
    Composite ⇑ (component / components)
END G-Composite.
```

IV. Application to Security Design Patterns

In this section, we apply the proposed approach to security design patterns to demonstrate the style of design space definition and pattern specification in the proposed approach.

A. The Design Space of Security Systems

Computer and network security replies on a wide range of issues and various levels. Here, as an example, we focus on the logic and context level of enterprise architecture. In this case, we can model security systems in box diagrams [14]. A box diagram consists of a number of boxes and arrows. Each box represents a subsystem or entity of the system. Each arrow represents a channel of information flow or interaction between subsystems. For the sake of space, we will only define the structural view of the design space. The dynamic view of system's behaviour will be omitted.

```
DESIGN SPACE SecuritySystems;
  TYPE
    Subsystem:
      name: STRING, content: [Value], description: [STRING];
    InfoFlow:
      name: STRING, from, to: Subsystem, type: [STRING];
  VIEW Structure;
      PROPERTY
          type: Subsystem -> {aataStore, computation};
          mode: Subsystem -> {active, passive};
      RELATION
          Is-a-part-of: Subsystem x Subsystem;
  END structure;
END SecuritySystems
```

B. Security System Design Patterns

Now, we demonstrate that security system design patterns can be design with a number of special components that fully fill various security specific functions, such as encryption and decryption.

Figure 1 shows the architecture of an indirect in-line authentication architecture, where *AI* stands for authentication information.

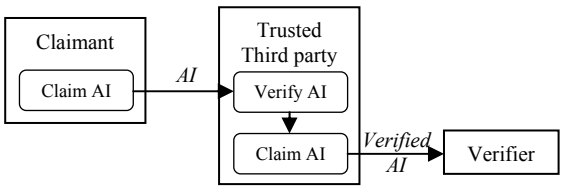

Figure 1. Indirect in-line authentication architecture

This architecture can be represented as follows.

```
PATTERN Indirect-In-Line-Authentication IN SecuritySystem;
  COMPONENT
    Claimant, TrustedThirdParty, Verifier: Subsystem;
    ClaimAI1, VerifyAI, ClaimAI2: Subsystem
    ClaimAI12VerifyAI, VerifyAI2ClaimAI2: InfoFlow;
    ClaimAI22Verifier: Infoflow;
  CONSTRAINT
    ClaimAI is-a-part-of Claimant;
    VerifyAI is-a-part-of TrustedThirdParty;
    ClaimAI2 is-a-part-of TrustedThirdParty;
    ClaimAI12VerifyAI.from = ClaimAI1;
    ClaimAI12VerifyAI.to = VerifyAI;
    VerifyAI2ClaimAI2.from= VerifyAI;
    VerifyAI2ClaimAI2.to = VerifyAI;
    ClaimAI22Verifier.from = ClaimAI2;
    ClaimAI22Verifier.to = Verifier;
END
```

An alternative authentication pattern is online authentication shown in Figure 2.

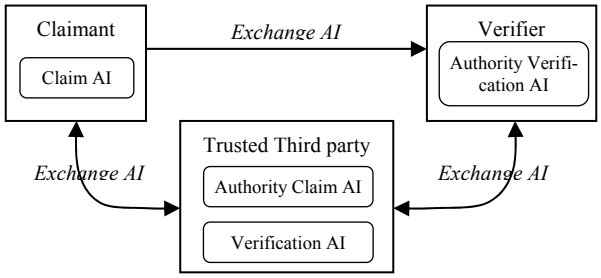

Figure 2. Online authentication architecture

```
PATTERN Online-Authentication IN SecuritySystem;
  COMPONENT
    Claimant, TrustedTP, Verifier: Subsystem;
    ClaimAI, AuthorityClaimAI, VerifAI: Subsystem;
    AuthorityVerifAI: Subsystem;
    ClaimantTrustedTP, VerifierTrustedTP: InfoFlow;
    ClaimantVerifier: InfoFlow;
  CONSTRAINT
    ClaimAI is-a-part-of Claimant;
    AuthorityClaimAI is-a-part-of TrustedTP;
    VerifAI  is-a-part-of TrustedTP;
    AuthorityVerifAI is-a-part-of Verifier;
      … (* Some constraints are omitted for the sake of space *)
END
```

Another set of examples of security design patterns are encryption and decryption techniques, as shown in Figure 3.

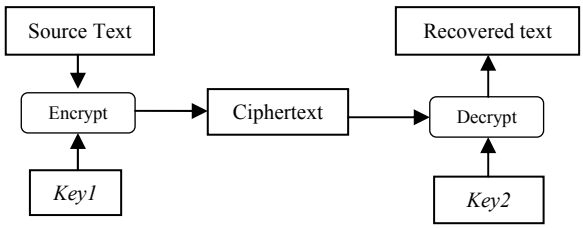

Figure 3. Encryption and decryption

```
PATTERN EncryptDecrypt IN SecuritySystem;
  COMPONENT
    encrypt, decrypt, source, ciphered, recovered,
      key1, key2:  Subsystem;
    source2encrypt, encrypt2ciphered, ciphered2decrypt,
    decrypt2recovered, key12encript, key22decrypt: InfoFlow;
```

```
  CONSTRAINT
    encrypt.type=computation; decrypt.type=computation;
    source.type=dataStore;   ciphered. type=dataStore;
    recovered. type=dataStore;
    key1.type=dataStore;  Key2.type=dataStore;
    source2encrypt.from=source;  source2encrypt.to= encrypt;
    encrypt2ciphered.from= encrypt;
    encrypt2ciphered.to= ciphered;
    ciphered2decrypt.from= ciphered;
    ciphered2decrypt.to= decrypt;
    decrypt2recovered.from= decrypt;
    decrypt2recovered.to= recovered;
    …
END
```

There are two types of encryption/decryption techniques: symmetric and asymmetric. The former uses the same key in encryption and decryption, while the later uses different keys. Thus, we have two specialisations of the patterns.

```
PATTERN SymetricEnDEcryppt in SecuritySystem  EQUALS
    EncryptDecrypt [key1.content =key2.content]   END
PATTERN AsymetricEnDEcryppt in SecuritySystem EQUALS
    EncryptDecrypt [not (key1.content = key2.content)]  END
```

Figure 4 shows a conceptual model of access control subsystem [14]. It is in fact a design pattern for access control in enterprise systems.

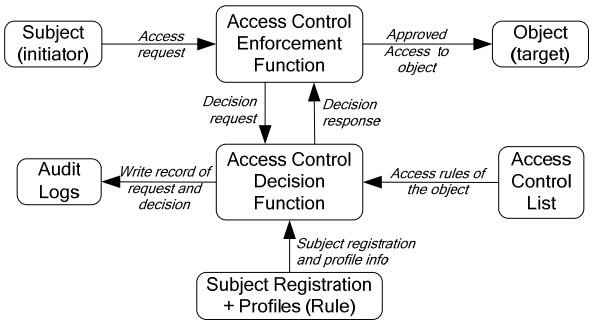

Figure 4. Conceptual model of access control system.

```
PATTERN AccessControl IN SecuritySystem
  COMPONENT
    Subject, EnforcementFun, DecisionFun, Object,
    AuditLogs, AccessControlList, SubjectReg: Subsystem;
    AccessReq, ApprovedAccessReq, DecisionReq,
    DecisionResp, WriteAuditRecord, SubjectInfo,
    AccessRule: InfoFlow;
  CONSTRAINT …
END.
```

V. Conclusion

In this paper we have proposed an approach to define design spaces so that design patterns in various subject domains can be defined in the same way as we define OO design patterns. We demonstrated the applicability of the proposed approach by examples of security design patterns. However, the structures of security systems have been simplified by representing them in box diagram models. Their dynamic features are omitted. The examples given in this paper are only skeletons. Many obvious constraints have been omitted for the sake of space. Further details must be worked out. There are also a number of other security design patterns can be identified. A case study of them and their composition is worth trying.

Existing research on relationships between DPs has limited to those within the same design space. However, to study patterns in cyberspaces, we need relationships between patterns across different design spaces. In particular, a security DP may be designated to against an attack pattern. They are in different design spaces. Hence, we have the following research questions:

- How to formally define the 'against' relationship between such pairs of patterns? And, how to prove a security pattern can successfully prevent all attacks (i.e. instances) of a certain attack pattern?
- Assume that the composition of security DPs (and attack patterns as well) be expressed in the same way as composition of OO DPs. Then, a question is: if a number of security patterns are composed together to enforce the security for an information system, can they prevent attacks of the target attack patterns and their all possible compositions?

Acknowledgement

The work reported in this paper is funded by the Oxford Brookes University. The author would like to thank Prof. David Duce, Dr. Ian Bayley and Mr. Clive Blackwell for discussions on related topics.

References

[1] Blakley, B. and Heath, C. *etc*. *Security Design Patterns*. Technical Guide, The Open Group, April 2004.

[2] Yoshioka, N, Washizaki, H, and Maruyama, K. A survey on security patterns. *Progress in Informatics* (5), pp35-47, 2008.

[3] Taibi, T. (eds), *Design patterns formalization techniques*, IGI Pub. 2007.

[4] Bayley, I. and Zhu, H. Formal Specification of the Variants and Behavioural Features of Design Patterns. *J. of Systems & Software* 83(2), pp 209–221, 2010,

[5] Bayley, I. and Zhu, H., A Formal Language for the Expression of Pattern Compositions. *Int'l J. on Advances in Software* 4(3&4), pp354 - 366, 2011.

[6] Zhu, H. and Bayley, I. An algebra of design patterns. *ACM Trans. on Software Eng. and Meth* (*in press*).

[7] Shaw, M., *The Role of Design Spaces*, IEEE Software, January/February 2012, 29(1), pp 46-50.

[8] Brooks F.P. Jr., *The Design of Design: Essays from a Computer Scientist*, Addison-Wesley, 2010.

[9] Yoshikawa, H. General design theory and a CAD system. *Proc. of the IFIP WG5.2-5.3 1980 Working Conference on Man-Machine Communication in CAD/ CAM*, pp35–57, North-Holland, 1981.

[10] Kakuda, Y. and Kikuchi, M. Abstract design theory. *Annals of Japan Ass. for Philosophy of Science*, 2001.

[11] OMG. *MDA Specification*. Object Management Group, USA, August 2010. URL: http://www.omg.org/mda/specs.htm.

[12] Zhu, H., An Institution Theory of Formal Meta-Modelling in Graphically Extended BNF. *Frontier of Computer Science* 6(1), pp40-56, 2012.

[13] Chiswell, I. and Hodges, W. *Mathematical Logic*, volume 3 of Oxford Texts in Logic. Oxford University Press, 2007.

[14] Sherwood, J., Clark, A. and Lynas, D. *Enterprise Security Architecture: A Business-Driven Approach*. CMP Books, 2005.