

Formal Modeling and Specification of Design Patterns Using RTPA

Yingxu Wang, University of Calgary, Canada

Jian Huang, University of Calgary, Canada

ABSTRACT

Software patterns are recognized as an ideal documentation of expert knowledge in software design and development. However, its formal model and semantics have not been generalized and matured. The traditional UML specifications and related formalization efforts cannot capture the essence of generic patterns precisely, understandably, and essentially. A generic mathematical model of patterns is presented in this article using real-time process algebra (RTPA). The formal model of patterns are more readable and highly generic, which can be used as the meta model to denote any design patterns deductively, and can be translated into code in programming languages by supporting tools. This work reveals that a pattern is a highly complicated and dynamic structure for software design encapsulation, because of its complex and flexible internal associations between multiple abstract classes and instantiations. The generic model of patterns is not only applicable to existing patterns' description and comprehension, but also useful for future patterns' identification and formalization.

Keywords: cognitive informatics; design patterns; formal method; pattern comprehension; pattern modeling; RTPA; software engineering; unified model of patterns

INTRODUCTION

Design patterns are a powerful tool for capturing software design notions and best practices, which provide common solutions to core problems in software development. Design patterns are a promising technique that extends reusability of software from code to design notions. A representative work of design patterns is initiated by Gamma and his colleagues in *Design Patterns: Elements of Reusable Object-Oriented*

Software in 1994 (Gamma, Helm, Johnson, & Vlissides, 1995). Design patterns may speed up the development process by providing tested and proven development paradigms. Reusing design patterns helps to prevent subtle issues in large-scale software development and improves code readability for architects and programmers. Design patterns can contribute to the definition, design, and documentation of class libraries and frameworks, offering elegant and reusable

solutions to design problems, and consequently increasing productivity and development quality (Gamma et al., 1995; Wang, 2007a). Each design pattern lets some aspects of the system structure vary independently of other aspects, thereby making the system more robust to a particular kind of change.

Design patterns are used to be modeled and specified in natural language narratives, object-oriented programming languages, and UML diagrams. The traditional means are either inherently ambiguous or inadequate (Lano, Goldsack, & Bicarregui, 1996; Vu & Wang, 2004; Wang & Huang, 2005). The major problems in current methodologies for pattern specification are identified as follows:

- **The lack of a unified and generic architecture of patterns as a multilayered complex entity with a set of abstract and concrete classes and their interrelations:** Patterns have been classified in three categories known as the *creational*, *structural*, and *behavioral* patterns (Gamma et al., 1995). However, the theories for the nature of patterns and their generic architecture are yet to be sought.
- **The lack of abstraction:** Almost all patterns are described as a specific and concrete case in natural language, UML diagrams, or some formal notations. However, no generic mathematical model of patterns is rigorously established, which may form a deductive basis for deriving concrete and application-specific patterns.
- **The lack of uniqueness:** In the conventional pattern framework, there are different patterns that may be implemented by similar code; Reversely, the same pattern may be implemented in various ways.
- **The use of unstructured semantic means to denote highly complicated design knowledge in patterns:** The informal descriptions of patterns puzzle users and cause substantial confusions. Even the creators of patterns demonstrate inconsistent over the semantics of certain patterns.

The authors perceive that the above fundamental problems can be alleviated by introducing formal semantics for design patterns and their generic mathematical models (Wang, 2002, 2003, 2006a-c, 2007a-c). This approach allows for unambiguous specifications, enables reasoning about the relationships between abstract and concrete patterns, and promotes a coherent framework for the rapidly growing body of software patterns in software engineering (Beck, Coplien, Crocker, & Dominick, 1996; Bosch, 1996; Wang, 2002, 2006a, 2007a). This article presents a generic model of design patterns and a formal specification method for design patterns using Real-Time Process Algebra (RTPA) (Wang, 2002, 2003, 2007a). The approach proposed in this article aimed at the following objectives:

- **It is generic:** The same pattern model can be adopted to specify any existing and future pattern, particularly user defined patterns. To some extent, the general pattern model is the pattern of patterns.
- **It is formalized:** The mathematical semantics and formal notation system are based on RTPA (Wang, 2002, 2003, 2007a).
- **It is expressive:** Only 34 notations are used to denote class association relationship and specify patterns from three facets known as the architecture, static behaviors, and dynamic behaviors of patterns.
- **It is structured:** Patterns are described from high-level to detailed-level via step-wise refinement using a coherent set of notations.

In this article, existing pattern specification techniques are reviewed in the following section. The RTPA methodology for pattern specification is introduced. A generic model of design patterns is rigorously modeled. Based on the mathematical semantics and general model, case studies on deriving specific pattern specifications are presented using three well known patterns such as the State, Strategy, and the MasterSlave patterns.

APPROACHES TO SOFTWARE PATTERN DESCRIPTION

A number of pattern modeling methodologies have been proposed, such as the layout object model (Bosch, 1996), the constraint diagrams (Lauder & Kent, 1998), the *language for pattern uniform specification* (Eden, Gil, Hirshfield & Yehudai, 2005), *meta-models* (Pagel & Winter, 1996; Sunye, Guennee, & Jezequel, 2000), *object calculus* (Lano et al., 1996), *pattern visualization techniques* (Lauder et al., 1998), and the *design pattern modeling language* (Mapelsden, Hosking, & Grundy et al., 1992). This section briefly reviews three major pattern specification methods and comparatively analyzes their strengths and weaknesses.

The Layout Object Model

The layout object model (LayOM) (Bosch, 1996) is an extension of object-oriented languages containing components that are not supported by the conventional object models such as layers, categories, and states. It supports the representation of design patterns in object-oriented programming languages.

In LayOM, layers are used to encapsulate objects and intercept messages that are sent to and by the objects. The layers are organized into classes and each layered class represents a concept, such as a relation with another object or a design pattern. A *state* in LayOM is a dimension of the abstract object state that is an externally visible abstraction of the internal, concrete state of objects. A *category* is defined as a client category that describes the discriminating characteristics of a subset of possible

clients. *Relations* in LayOM are denoted by structural, behavioral, and application-domain relations.

For example, the *adapter* design pattern can be described in LayOM, as shown in Figure 1, which converts the interface of a class into another interface that is expected by its clients. The adapter layer can be used to class adaptation by defining a new adapter class consisting only of two layers.

In LayOM, a one-to-one mapping between the design and implementation of a pattern is provided. Another advantage of it is that there is no requirement for defining a method for every method that needs to be adapted. However, a disadvantage of the adapter technique is that the arguments of the message will be passed as is, which is not flexible to cover semantic analyses. Another drawback is the implementation overhead, because messages sent to or from an object need to pass all the defined layers.

The Constraint Diagrams

The *constraint diagram* (CD) (Lauder & Kent, 1998) denotes a pattern by three separate models known as the role, type, and class. A *role* model in CD is the most abstract and depicts layer that describes the essential spirit of a pattern without specific details. A *type* model in CD refines to the role model with abstract states and operation interfaces forming a domain-specific refinement of the pattern. A *class* in CD model implements the type model, thus deploying the underlying pattern in terms of concrete classes.

An *abstract factory pattern* deployed as a constraint model is shown in Figure 2. The core

Figure 1. The adapter pattern described in LayOM (Bosch, 1996)

```

class Adapter
  layers
  adapt: Adapter (accept mess1 as newMessA,
    accept mess2 and mess3 as newMessB);
  inh: Inherit (Adaptee);
  end;

```

of the pattern is represented as a role mode, further refined by a type model, and implemented by a class model. However, a graphical model is not enough for a precise and unambiguous specification. There is still a need to describe additional constraints about the objects in the model. Otherwise, ambiguities cannot be avoided in the model (OMG, 1997).

LePUS: Language for Patterns Uniform Specification

Language for patterns uniform specification (LePUS) is a declarative language based on logic introduced in 1997 (Eden et al., 2005). It models class relationships and semantics, and facilitates reasoning with higher order sets. LePUS permits concise description of complex software artifacts. In LePUS, a program is represented primarily as a set of ground entities and relationships among them. Various interactions and associations that occur between participants of design patterns are abstracted classes and functions. A uniform set of classes of a dimension d is denoted by the class of dimension $d+1$.

Total relations are functions that describe the relations between two sets of entities. Bijective and regular correlations between sets of functions, classes, and hierarchies may also be modeled. A specification of the *strategy* pattern in LePUS is shown in Figure 3.

The advantage of LePUS is its higher order logic means. However, it is difficult to directly map a LePUS specification of patterns into executable programs. In addition, patterns specified in LePUS are application specific rather than generic.

Other Approaches

Meta-models for design pattern instantiations and validations are proposed in Pagel and Winter (1996) and Sunye et al. (2000) without supporting for code generation. In Florijn, Meijers, and Wionsen (1997), the *fragment-based technique* allows the representation and composition of design patterns. A design pattern proof techniques is proposed in Lano et al. (1996) using the *object calculus* as a semantic framework. The *design pattern modeling language* (DPML) is proposed in Mapelsden et

Figure 2. The constrain diagram of an abstract factory pattern (Lauder & Kent, 1998)

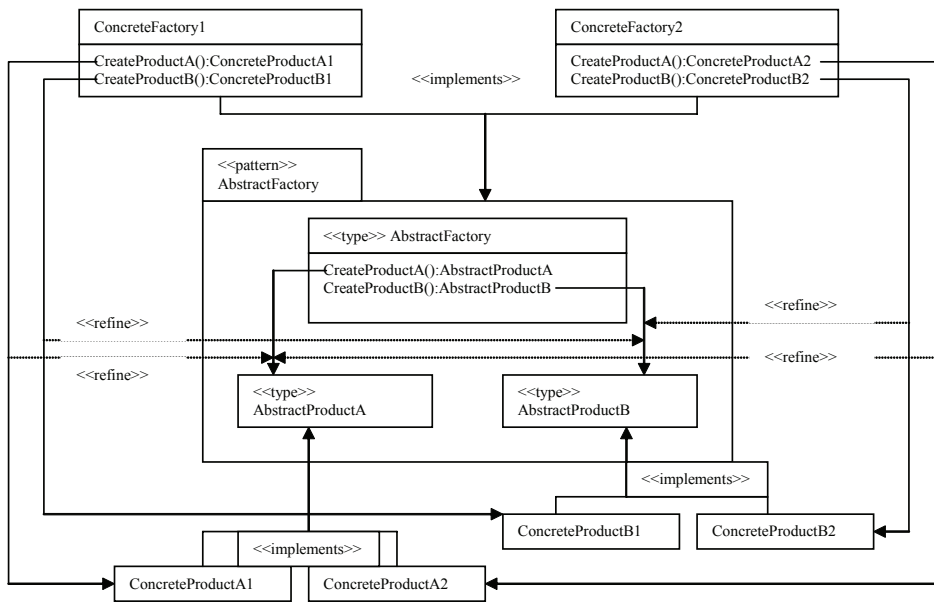


Figure 3. The LePUS Specification of the strategy pattern (Eden et al., 2005)

$\begin{aligned} &\exists \text{context}, \text{client} \in C \\ &\text{operations} \in F \\ &\text{Algorithm}, \text{Configure} - \text{Contexts} \in 2^F. \\ &\text{Strategies} \in H \\ &\text{clan}(\text{operation}, \text{context}) \wedge \\ &\text{clan}(\text{Algorithm}, \text{Strategies}) \wedge \\ &\text{tribe}(\text{Configure} - \text{Context}, \text{client}) \wedge \\ &\text{Invocation}^{\rightarrow}(\text{operation}, \text{Configure} - \text{Context}) \wedge \\ &\text{Invocation}^{\rightarrow}(\text{Algorithm}, \text{context}) \wedge \\ &\text{Argument} - \text{I}^{\rightarrow}(\text{Algorithm}, \text{context}) \wedge \\ &\text{Creation}^{\rightarrow H}(\text{Configure} - \text{Context}, \text{Strategies}) \wedge \\ &\text{Assignment}^{\rightarrow H}(\text{Context}, \text{Configure} - \text{Context}, \text{Strategies}) \wedge \\ &\text{Reference} - \text{to} - \text{Single}^{\leftarrow}(\text{context}, \text{Strategies}) \end{aligned}$

al. (1992) as a visual language for modeling patterns and their instances.

A common weakness of the methods proposed so far is that they concentrate only on specific pattern descriptions. The essence of pattern structures and the generic pattern theory are overlooked. Therefore, there is still a need to seek a more powerful means and methodology that help users to utilize pattern theories and models freely in pattern-based system design.

THE RTPA METHODOLOGY FOR PATTERN MODELING AND SPECIFICATION

In the preceding section, it can be seen that existing methods for pattern specifications were inadequate and inefficient in generic pattern specifications. This section adopts RTPA to formally specify design patterns, which is a set of new mathematical notations for formally describing system architectures, static and dynamic behaviors (Wang, 2002, 2006b).

RTPA is a mathematic-based software engineering notation system and a formal method for addressing problems in software system specification, refinement, and implementation. RTPA provides an expressive means for the formal and explicit description of software patterns in order to enhance the readability of pattern architecture, semantics, and behaviors. In RTPA, a software system is perceived and

described mathematically as a set of coherent processes. RTPA encompasses 17 meta processes and 17 algebraic process combination rules known as the process relations. Detailed description of RTPA may be referred to Wang (2002, 2003, 2007a).

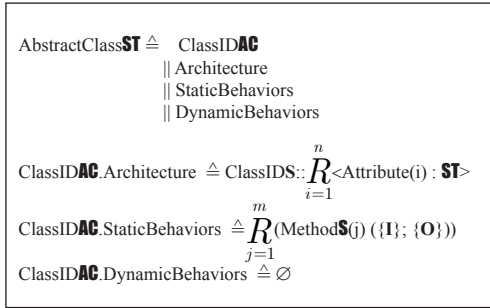
The Generic Model of Classes in RTPA

A unified notion of classes can be formally described by RTPA as given in “Wang (2007a).” The fundamental types of classes are the Abstract Classes (ACs) and the Concrete Classes (CCs). The former is a class that serves as a general and conceptual model to be inherited but not be instantiated. The latter is an ordinary class derived from an AC that can be instantiated.

A generic AC specified in RTPA is shown in Figure 4. The architecture part is used to specify the architectural attributes of the abstract class, which include internal variables shared by the hierarchy of classes. The static behavior part of an AC is used to define the method signatures of the AC class, where detailed implementation will be left to be done in derived concrete classes. It is noteworthy that an AC has no dynamic behavior in the specification because ACs cannot be instantiated.

Similarly, a generic CC can be formally specified in RTPA, as shown in Figure 5. The CC implements the dynamic behaviors that can be instantiated and executed in a derived

Figure 4. Abstract class specification in RTPA

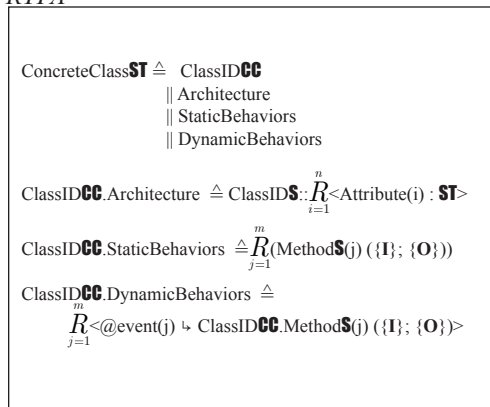


object. Possible events that drive a method in a **CC** can be classified into message, time, and interrupt. More formal treatment of classes and their relational operations may be referred to RTPA (Wang, 2002, 2003, 2007a) and concept algebra (Wang, 2006a, 2007c).

The Generic Model of Patterns in RTPA

A pattern is a highly reusable and coherent set of complex classes that are encapsulated to provide certain functions. According to “Gamma et al. (1995),” patterns can be classified into the *creational*, *structural*, and *behavioral* ones. Using RTPA notations and methodology, a pattern is denoted by three parallel components known as

Figure 5. Concrete class specification in RTPA



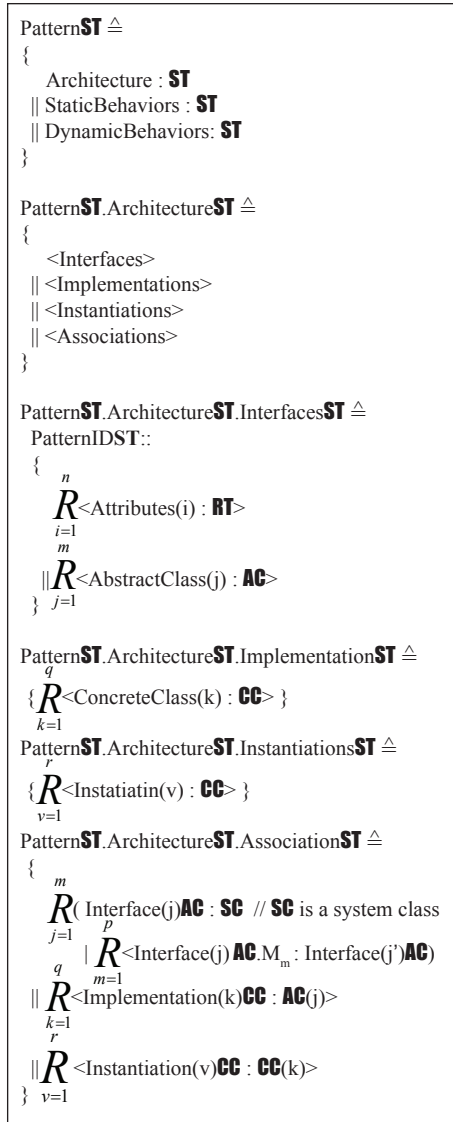
the *architecture*, *static*, and *dynamic* behaviors, as shown in Figure 6. The architecture of a pattern specifies how many classes and components are used to compose the pattern and what relationships are among those components. The static behaviors of a pattern define what kinds of components are used to compose this pattern and what rules all components should abide. The dynamic behaviors of a pattern describe how those components interact and collaborate to realize functionality at run-time.

In Figure 6, the *architecture* of the generic pattern can be refined by a set of component logic models (CLMs), which describes the structures of a class, particularly its attributes (Wang, 2002). The *static behaviors* of the generic pattern are refined by a set of processes that are corresponding to each of the methods within the class. The process behaviors are denoted by RTPA meta-processes and their combinations using process relations for manipulating internal attributes or interacting with external components of the generic pattern. The *dynamic behaviors* of the generic pattern are refined by event-driven processes deployed by the system.

The generic pattern model may be treated as a super metapattern in object-oriented system design and programming, which models any specific software pattern at four levels known as the *interface*, *implementation*, *instantiations*, and *associations*. According to the generic model of patterns, the features of patterns lie in the hierarchical architectures as described by Pattern**ST**.Architecture**ST**, as shown in Figure 6. It is noteworthy that a class is usually modeled as a two-level structure with only the class *interface* and *implementations* in literature (Taibi, & Ngo, 2003; Vu & Wang, 2004). However, the four-level hierarchical model introduced here reveals the nature of how classes may be used to form complex patterns via *instantiations* and *associations*.

The *interface* of a pattern, Pattern**ST**.Architecture**ST**.Interface**ST**, isolates users of the pattern from its internal implementation. Users may only access the pattern via its interface. This mechanism enables the implementation of the

Figure 6. The generic mathematical model of design patterns (Wang, 2007a)



pattern independent of its users. Whenever the internal implementation needs to be changed, it is transparent to the users of the pattern as long as the interface keeps the same (Wang & Huang, 2005). The interface of a pattern specifies the communication protocol among pattern components. Although instances could extend

their behaviors beside those interface defined in this part their communication should abide those definition. The interface is the only access point of a component inside a pattern.

Because a pattern is a highly reusable construct of a software entity, the *implementation* of a pattern, PatternST.ArchitectureST.ImplementationST, is kept at a generic abstract class until the pattern is invoked by a specific application or instantiation. In other words, because a pattern is a generic model of reusable functions, specific behaviors in an execution instance are dependent on run-time information provided by users of the pattern.

The fourth component in the generic pattern hierarchy is the internal *associations*, which is used to model the interrelationships among other three-level abstractions of classes and interfaces within the pattern. The *associations* of the pattern define relationships among all components in the pattern. Component collaborations are the soul of patterns that capture component collaborations. The flexibility, reusability, and differences of patterns are embodied by the associations (Wang & Huang, 2005). A comprehensive set of pattern association rules may be referred to Concept Algebra (Wang, 2006b, 2007c).

The generic model of patterns or the meta pattern model describes software patterns in a coherent, concise, and unambiguous way. External relationship among patterns could be deduced by this formula as well, when a super pattern is considered beyond all the component patterns. The method developed in this section helps readers avoid the drawbacks of conventional patterns, in order to develop more efficient, reusable, flexible, and predicable software systems.

CASE STUDIES ON FORMAL SPECIFICATIONS OF PATTERNS IN RTPA

This section applies the RTPA pattern specification methodology in three case studies. The *State* and *Strategy* patterns are used to demonstrate that differences between these two patterns can

be clearly distinguished by RTPA specifications, while other methods rendering vague message to users. The *MasterSlave* pattern (Buschmann, 1995) is used to demonstrate the expressive power of RTPA specifications. All cases studies show that not only conventional design patterns, but also newly discovered patterns can be precisely specified by RTPA.

Formal Specification of the State Pattern

The *State* pattern allows an object to alter its behavior when its internal state changes. The structure of this pattern proposed in Gamma et al. (1995) is shown in Figure 7.

A formal model of the state pattern can be derived on the basis of the generic pattern model developed in Figure 6. Corresponding to Figure 7, the RTPA specification of the *State* pattern is given in Figure 8.

Formal Specification of the Strategy Pattern

The *Strategy* pattern defines a family of algorithms, encapsulates them in a coherent structure, and makes them interchangeable (Gamma et al., 1995). This pattern lets the algorithm vary independently from clients that use it. The structure of the Strategy pattern is shown in Figure 9.

A formal model of the Strategy pattern can be derived on the basis of the generic pattern model developed in Figure 6. Corresponding to Figure 9, the RTPA specification of the strategy pattern is shown in Figure 10.

Contrasting the State and Strategy patterns in UML, it is noteworthy that both patterns share almost identical structures. In other words, UML class diagrams, as shown in Figures 7 and 9, may not be able to discriminate the differences between these two patterns. This results in the main confusions, ambiguities, and difficulties in pattern comprehension and applications using UML-based methodologies.

However, the specifications in RTPA, as shown in Figures 8 and 10, can clearly capture the differences in the sections of class associations by different composition approaches. The RTPA models show that in the former, all concrete state classes are instantiated by concrete classes simultaneously, where the concrete state objects have the same lifecycle as those of the concrete context objects; while in the latter, only one concrete strategy object alive within any point of time in the concrete context object lifecycle. This is the essential difference between those two patterns, which cannot be expressed explicitly by UML syntaxes and semantics.

Formal Specification of the MasterSlave Pattern

It is observed that traditional methods for pattern modeling are focused on existing and specific patterns proposed in "Gamma et al. (1995)." However, most practical patterns in software engineering are user-defined rather than pre-specified. Therefore, a generic pattern model is needed to support users to deductively model new patterns in practice.

Figure 7. The UML structure of the State pattern

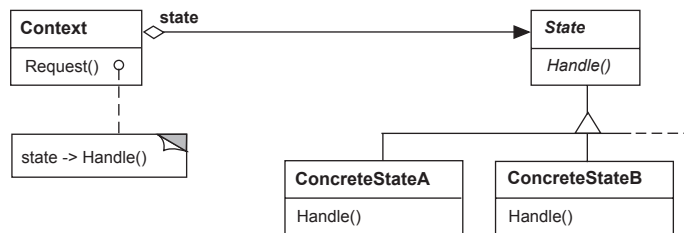
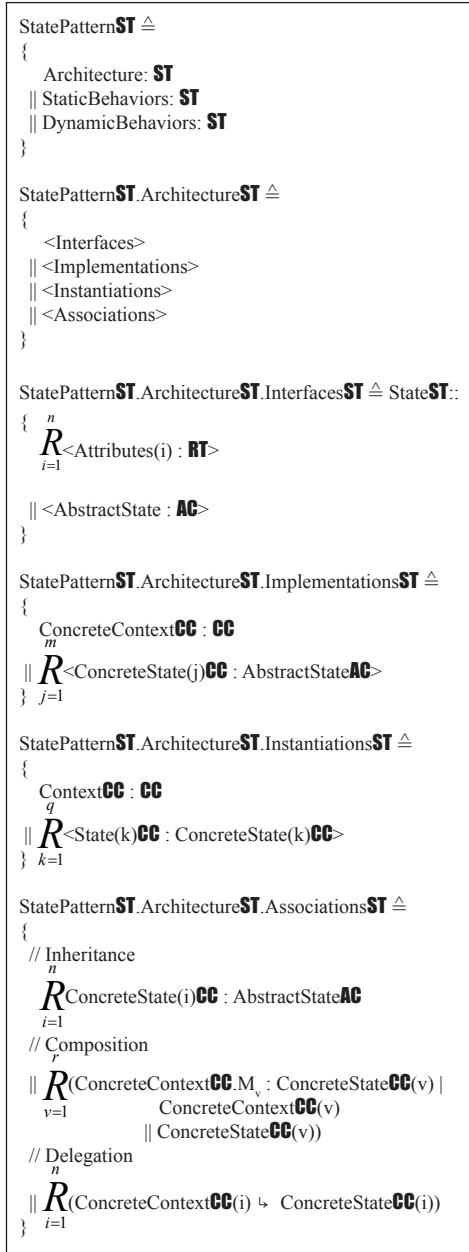
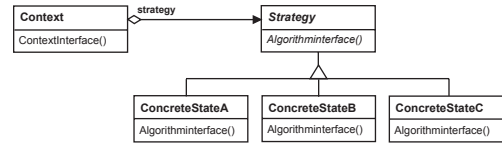


Figure 8. The RTPA specification of the State pattern



A system pattern, *MasterSlave* (Buschmann, 1995) as shown in Figure 11, is presented in this subsection in order to demonstrate the application of the generic pattern model

Figure 9. The UML structure of the Strategy pattern



and the expressive power of RTPA for modeling patterns. The *MasterSlave* pattern handles the computation of replicated services of a software system to achieve fault tolerance and robustness. Replication of services and the delegation of the same task to several independent slave servers is a common strategy to handle fault-tolerant requirements in safety-critical software systems.

The *MasterSlave* pattern consists of two kinds of components: the master and the slaves. Clients of the pattern interact with the master component directly. However, the master component does not implement services by itself. It delegates the services to a number of slave components, where at least two identical slave components exist in the system with the same set of functionality. The slave components are completely independent of each other, and they may use different strategies for providing the designated service. The master component delegates a requested service to all slave components and chooses one of the most suitable responses as the result for the client.

A formal model of the *MasterSlave* pattern can be derived on the basis of the generic pattern model developed in Figure 6. Corresponding to Figure 11, the RTPA specification of the *MasterSlave* pattern is given in Figure 12. The derived pattern precisely describes the architecture and associations between member classes of the *MasterSlave* pattern. Several strategies may be available for the master component to select results provided by the slaves, such as the result first returned, the majority result returned by all slaves, or the average result of all slaves.

Figure 10. The RTPA specification of the Strategy pattern

```

StrategyPatternST  $\triangle$ 
{
  Architecture: ST
  || StaticBehaviors: ST
  || DynamicBehaviors: ST
}

StrategyPatternST.ArchitectureST  $\triangle$ 
{
  <Interfaces>
  || <Implementations>
  || <Instantiations>
  || <Associations>
}

StrategyPatternST.ArchitectureST.InterfacesST  $\triangle$  StrategyST
::
{
   $\prod_{i=1}^n R$ <Attributes(i) : RT>
  || <AbstractStrategy : AC>
}

StrategyPatternST.ArchitectureST.ImplementationsST  $\triangle$ 
{
  ConcreteContextCC : CC
   $\prod_{j=1}^m R$ (ConcreteStrategy(j)CC : AbstractStrategyAC)
}

StrategyPatternST.ArchitectureST.InstantiationsST  $\triangle$ 
{
  ContextCC : CC
   $\prod_{k=1}^q R$ <Strategy(k)CC : ConcreteStrategy(k)CC>
}

StrategyPatternST.ArchitectureST.AssociationsST  $\triangle$ 
{
  // Inheritance
   $\prod_{i=1}^n R$ ConcreteStrategy(i)CC : AbstractStrategyAC
  // Composition
  || ( ConcreteContextCC.M : ConcreteStrategyCC |
        ConcreteContextCC
        || ConcreteStrategyCC )
  // Delegation
  || ConcreteContextCC  $\hookrightarrow$  ConcreteStrategyCC
}
    
```

Figure 11. The UML structure of the Master-Slave pattern

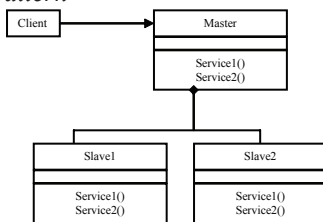


Figure 12. The RTPA specification of the Master-Slave pattern

```

MasterSlavePatternST  $\triangle$ 
{
  Architecture : ST
  || StaticBehaviors : ST
  || DynamicBehaviors : ST
}

MasterSlavePatternST.ArchitectureST  $\triangle$ 
{
  <Interfaces>
  || <Implementations>
  || <Instantiations>
  || <Associations>
}

MasterSlavePatternST.ArchitectureST.InterfacesST  $\triangle$ 
MasterST ::
{
   $\prod_{i=1}^n R$ <Attributes(i) : RT>
  || <AbstractMasterComponent : AC>
}

MasterSlavePatternST.ArchitectureST.ImplementationsST  $\triangle$ 
{
  <ConcreteMasterComponentCC : CC>
   $\prod_{j=1}^m R$ <ConcreteSlaveComponent(j)CC : CC>
}

MasterSlavePatternST.ArchitectureST.InstantiationsST  $\triangle$ 
{
  ConcreteMasterInstanceCC : AbstractMasterComponentAC
   $\prod_{k=1}^q R$ <ConcreteSlaveInstance(k)CC :
    ConcreteSlaveComponent(j)CC>
}

MasterSlavePatternST.ArchitectureST.AssociationsST  $\triangle$ 
{
  // Inheritance
  ConcreteMasterComponentCC :
    AbstractMasterComponentAC
  // Delegation
  || ( ConcreteMasterComponentCC  $\hookrightarrow$ 
         $\prod_{j=1}^m R$ ConcreteSlaveComponent(j)CC )
  // Aggregation
  || (  $\prod_{k=1}^q R$ ConcreteMasterComponentCC.M :
        ConcreteSlaveComponent(k) |
        ConcreteMasterComponentCC
        || ConcreteSlaveComponentCC )
}
    
```

CONCLUSION

This article has reviewed existing pattern specification methods and problems yet to be solved. A generic mathematical model of patterns has been presented using Real-Time Process Algebra (RTPA). Based on it any design patterns, either system-specified or user-defined, can be derived. With the RTPA support tool, a pattern specified in RTPA can be automatically translated into code in programming languages (Tan, Wang, & Ngolah, 2006).

This work has revealed that a software pattern is a highly reusable design encapsulation that encompasses complex and flexible internal associations between a coherent set of abstract classes and instantiations. The generic model of patterns has provided a pattern of patterns. It is not only applicable to existing patterns' modeling and comprehension, but also useful for future patterns' identification and formalization.

ACKNOWLEDGMENT

The authors would like to acknowledge the Natural Science and Engineering Council of Canada (NSERC) for its partial support to this work. We would like to thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

- Beck, K., Coplien, J. O., Crocker, R., & Dominick, L. (1996, March). Industrial experience with design patterns. In *Proceedings of the 19th Intel. Conf. on Software Engineering*, (pp. 103-114). Berlin: IEEE CS Press.
- Bosch, J. (1996). Relations as object model components. *Journal of Programming Languages*, 4(1), 39-61.
- Buschmann, F. (1995). *The MasterSlave Pattern, pattern languages of program design*. Addison-Wesley.
- Eden, A. H., Gil, J., Hirshfeld, Y., & Yehudai, A. (2005). *Towards a mathematical foundation for design patterns* (Tech. Rep.). Dept. of Computer Science, Concordia University, Montreal, Canada.
- Florijn, G., Meijers, M., & Wionsen, P. V. (1997). Tool support for object-oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*(pp. 472-495), Jyvaskyla, Finland.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object oriented software*. Reading, MA: Addison-Wesley.
- Lano, K., Goldsack, S., & Bicarregui, J. (1996). Formalizing design patterns. In *Proceedings of the 1st BCS-FACS Northern Formal Methods Workshop* (p. 1).
- Lauder, A., & Kent, S. (1998). Precise visual specification of design patterns. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, (LNCS, 1445, pp. 114-134). Springer-Verlag.
- Mapelsden, D., Hosking, J., & Grundy, J. (1992). *Design pattern modeling and instantiation using DPML*, (Tech. Rep.). Department of Computer Science, University of Auckland.
- OMG. (1997). *Object Constraint Language Specification 1.1*.
- Pagel, B. U., & Winter, M. (1996). Towards pattern-based tools. In *Proceedings of the EuropLop'96*, (pp. 3.1-3.11).
- Sunye, G., Guennec, A. L., & Jezequel, J. M. (2000). Design patterns application in UML. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*(pp. 44-62), Sophia Antipolis, France.
- Taibi, T., & Ngo, D. C. L. (2003). Formal specification of design patterns—a balanced approach. *Journal of Object Technology*, 2(4), 127-140.
- Tan, X., Wang, Y., & Ngolah, C. F. (2006, May). Design and implementation of an automatic RTPA code generator. In *Proceedings of the 2006 Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, (pp. 1605-1608). Ottawa, Canada: IEEE CS Press.
- Vu, N. C., & Wang, Y. (2004, May). Specification of design patterns using real-time process algebra (RTPA). In *Proceedings of the 2004 Canadian Conference on Electrical and Computer Engineering (CCECE'04)*, (pp. 1545-1548). Niagara, Falls, Ontario: IEEE CS Press.

- Wang, Y. (2002, October). The real-time process algebra (RTPA). *Annals of Software Engineering: An International Journal*, 14, 235-274.
- Wang, Y. (2003). Using process algebra to describe human and software behaviors. *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy*, 4(2), 199-213.
- Wang, Y. (2006a, July). On concept algebra and knowledge representation. In *Proceedings of the 5th IEEE International Conference on Cognitive Informatics (ICCI'06)*, (pp. 320-331). Beijing, China: IEEE CS Press.
- Wang, Y. (2006b). On the informatics laws and deductive semantics of software. *IEEE Transactions on Systems, Man, and Cybernetics (C)*, 36(2), 167-171.
- Wang, Y. (2006c, July). Cognitive informatics and contemporary mathematics for knowledge representation and manipulation, Invited Plenary Talk. In *Proceedings of the 1st International Conference on Rough Set and Knowledge Technology (RSKT'06)*, (pp. 69-78). Lecture Notes in Artificial Intelligence, LNAI 4062. Chongqing, China: Springer-Verlag.
- Wang, Y. (2007a, July). *Software engineering foundations: A software science perspective*. CRC Book Series in Software Engineering (Vol. II). USA: CRC Press.
- Wang, Y. (2007b, January). The theoretical framework of cognitive informatics. *The International Journal of Cognitive Informatics and Natural Intelligence (IJCiNi)*, 1(1), 1-27. Hershey, PA: IGI Publishing.
- Wang, Y. (2007c). Keynote speech, on theoretical foundations of software engineering and denotational mathematics. In *Proceedings of the 5th Asian Workshop on Foundations of Software*, Xiamen, China, (pp. 99-102).
- Wang, Y., & Huang, J. (2005, May). Formal models of object-oriented patterns using RTPA. In *Proceedings of the 2005 Canadian Conference on Electrical and Computer Engineering (CCECE'05)*, Saskatoon, Canada, (pp. 1822-1825). IEEE CS Press.

Yingxu Wang is professor of cognitive informatics and software engineering, director of the International Center for Cognitive Informatics (ICfCI), and director of the Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He received a PhD in software engineering from The Nottingham Trent University, UK, in 1997, and a BSc in electrical engineering from Shanghai Tiedao University in 1983. He was a visiting professor in the computing laboratory at Oxford University during 1995, and has been a full professor since 1994. He is editor-in-chief of International Journal of Cognitive Informatics and Natural Intelligence (IJCINI), editor-in-chief of the IGI book series of Advances in Cognitive Informatics and Natural Intelligence, and editor of CRC book series in Software Engineering. He has published over 300 papers and 10 books in software engineering and cognitive informatics, and won dozens of research achievement, best paper, and teaching awards in the last 28 years, particularly the IBC 21st Century Award for Achievement "in recognition of outstanding contribution in the field of Cognitive Informatics and Software Science."

Jian Huang is a masters degree candidate with Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He received a BSc degree in computer science from Jiangsu University, China in 1991. He also received a System Analyst Certificate in software engineering since 1993. His research area is in formal methods for software pattern specifications.