JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 4, July-August 2003

Formal Specification of Design Patterns - A Balanced Approach

Toufik Taibi, Multimedia University, Cyberjaya, Malaysia
David Chek Ling Ngo, Multimedia University, Melaka, Malaysia

Abstract

Pattern users are faced with difficulties in understanding when and how to use the increasing number of available design patterns. This is mainly due to the inherent ambiguity in the existing means (textual and graphical) of describing them. Hence, there is a need to introduce formalism in order to describe them accurately and allow rigorous reasoning about them. The main problem of existing formal specification languages for design patterns is their lack of completeness. This is mainly because they tend to focus on specifying either the structural or the behavioral aspect of design patterns but not both of them. We propose a simple yet Balanced Pattern Specification Language (BPSL) that is aimed to achieve equilibrium by specifying both aspects of design patterns. BPSL combines two subsets of logic, one from First Order Logic (FOL) and one from Temporal Logic of Actions (TLA).

1 INTRODUCTION

Design patterns are abstractions generated from the valuable experiences of developers in solving problems repeatedly encountered within certain contexts. Since design patterns have been extensively tested and used in many development efforts, reusing them yields better quality software within a reduced time frame.

Pioneer pattern writers needed an urgent means to describe these cumulated experiences in order to allow developers to reuse them. At the early stage of pattern evolution, a combination of textual descriptions, OO graphical modeling languages and sample code fragments was sufficient for conveying the essence behind patterns. Initial efforts were focused on building a pattern vocabulary, a community of pattern writers and users, and a pattern literature. At a later stage, it has been found that patterns cannot be used in isolation from other patterns but as micro-architectures that when combined together can solve a component or even the whole system.

However, as the number of patterns has grown, and problems requiring combining patterns surfaced, users started to realize that textual description can be ambiguous and sometimes misleading in understanding and applying patterns. Unsettled debates were

Cite this article as follows: Toufik Taibi, David Check Ling Ngo: "Formal Specification of Design Patterns – A Balanced Approach", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 127-140. http://www.jot.fm/issues/issue_2003_07/article4

initiated among users and even pattern writers themselves on various aspects of patterns [Vliss97a][Vliss97b].

Hence, there was a need for a formal means of accurately describing design patterns. Formal specification of design patterns is not meant to replace the existing textual/graphical descriptions but rather to complement them to achieve well-defined semantics, allow rigorous reasoning about them and facilitate tool support.

Formal specification of design patterns can enhance the understanding of their semantics. It can be used to help pattern users decide which pattern(s) is (are) more appropriate to solve a given design problem within a context. It can help formalize the combination of design patterns. Finally it can facilitate the development of tools for finding instances of patterns in programs and fine-tuning them to meet pattern specification [Eden01].

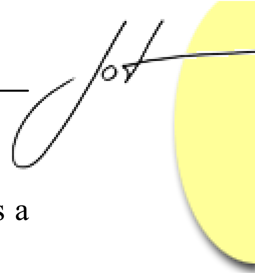
As the pattern field has matured, a number of formal specification languages [Chinn99] have emerged to cope with the inherent shortcomings of textual and graphical descriptions. However, since these specification languages originated from different mathematical sources and incorporated different ingredients, they reflect the way their authors perceived *"how should patterns be formalized?"* Their main problem is lack of completeness. This is mainly because they were not originally meant to specify design patterns and have been adapted to do so, or because they focused on specifying either the structural or behavioral aspect of design patterns but not both of them.

In this paper, we propose a Balanced Pattern Specification Language (BPSL) that is meant to accurately convey the essence of patterns in a balanced way. In [Taibi01], we described why and how should patterns be formalized and concluded that combining the formal specification of structural and behavioral aspects of design patterns in one specification helps specify patterns in a balanced way. BPSL uses First Order Logic (FOL) to specify the structural aspect and Temporal Logic of Actions (TLA) to specify the behavioral aspect. Although we approach the formalization in terms of specific examples, the underlying principles are applicable to any design pattern.

The rest of the paper is organized as follows. In the next section we lay down the foundations on top of which BPSL is built. Section 3 gives a detailed description of BPSL, while in section 4 we apply BPSL to the *Observer* design pattern [Gamma95], which has a significant behavioral aspect. Finally, in section 5 we present work related to what is presented in this paper and conclude the paper in section 6.

2 SETTING THE SCENE

Most attempts of formalizing design patterns have focused on the solution part, which is primarily defined using the structure, participants and collaboration sections of the pattern description [Gamma95]. Focusing on the solution part does not mean ignoring the other aspects of a pattern such as the problem (defined by its forces) and the context. The solution is the most tangible aspect of a pattern that can be easily translated to some sort



of formalism that would facilitate the understanding and usage of the design pattern as a whole.

Moreover, at this stage of pattern evolution, formalizing the other aspects of a pattern will not add any thing to the existing textual descriptions. As such, in the remaining of this document when we refer to a design pattern we mean its solution part. Moreover, we will use design patterns and patterns interchangeably as BPSL is mainly meant to formally specify design patterns not other types of patterns such as analysis, architecture, organizational, etc.

Design patterns differ in terms of their field of usage, the problems they solve and their context. However each pattern has a structural aspect and a behavioral aspect. Hence, any formal specification that is claimed to completely describe design patterns should incorporate the specification of both structural and behavioral aspects [Taibi01]. Each pattern can be seen from two complementary views: the structural view and the behavioral view. By balanced (in BPSL) we mean that the formal specification of both the structural and behavioral aspect of patterns should complement each other.

As BPSL is aimed to describe patterns accurately and in a balanced manner through a simple and concise notation, its main target is pattern understandability, which can only be achieved by understanding a pattern's structural and behavioral aspects and how they complement each other. By doing so, users will be able to know when and how to use a given pattern, which is crucial to taking full advantage of the inherent benefits of patterns.

The structural aspect of a design pattern can be formalized using a subset of First Order Logic (FOL), because relations between pattern participants can be easily expressed as predicates. For simplicity, the subset of FOL used focuses on variable symbols and predicate symbols. The behavioral aspect of a design pattern can be formalized using a subset of Temporal Logic of Actions (TLA) [Lamport94], which is best suited to describe collective behavior, i.e. how objects cooperate. The subset used focuses on actions that change state variables (class attributes) and/or associate or disassociate object through temporal relations. BPSL integrates two subsets of logic, one fraction of FOL and one from TLA in an attempt to describe patterns in an accurate and balanced way.

Following are the building blocks of BPSL. They reflect entities (participants) and relations (collaborations) between them in a design pattern.

1. Classes, attributes, methods, objects, and untyped values make the *primary* entities, which are considered irreducible units. Untyped values are values of variables of any type. They are used as a construct at a higher level of abstraction as opposed to low level programming language constructs. Untyped values and objects are used as parameters to actions (see section 3).
2. Relations express the way entities collaborate. Relations can either be permanent or temporal. Permanent relations between entities once defined cannot be changed while temporal relations may change throughout the execution of actions. BPSL

defines a set of primary permanent relations based on which other permanent relations can be built (see Table 1).

3. Actions are atomic units of execution, which can be understood as multi-object methods used to embody the behavioral aspect of design patterns. Actions associate and disassociate objects through temporal relations.
4. Any newly defined entity or permanent relation must derive from the *primary* entities and *primary* permanent relations respectively.

3 BALANCED PATTERN SPECIFICATION LANGUAGE (BPSL)

In this section we will take a deep and closer look at the BPSL building blocks (entities, relations and actions) that have been defined in the previous section. This section is divided into three subsections. The first describes how the structural aspect of design patterns is formalized using BPSL. The second section focuses on the specification of the behavioral aspect of design patterns. Finally, the last section shows how these two aspects when combined complement each other.

Structural Aspect

The subset of FOL used to describe the structural aspect of design patterns consists of variable symbols, connectives (mainly $\wedge$), quantifiers (mainly $\exists$) and predicate symbols acting upon variable symbols. Variable symbols represent classes, attributes, methods, objects and untyped values while the predicate symbols represent permanent relations. The domain (set) of *primary* entities that are classes, attributes, methods, objects, and untyped values is designated receptively ***C, A**, **M, O,** and **V***.

Table 1 depicts the *primary* permanent relations, their domain and their intent. These relations straightforwardly derive from object-oriented technology concepts. It is the smallest set (in terms of number of elements) on top of which any other permanent relation can be built.

For example the permanent relation *Forwarding* is a special case of *Invocation*, where the actual arguments in the invocation are the formal arguments defined for the first method. This can be formally specified as follows: $Forwarding\ (m_1,m_2) \Leftrightarrow Invocation(m_1,m_2) \wedge Argument(a_1, m_1) \wedge \ldots \wedge Argument(a_n,m_1) \wedge Argument(a_1, m_2) \wedge \ldots \wedge Argument(a_n,m_2)$, where $m_1, m_2 \in M$ and $a_1, \ldots, a_n \in V \cup C$ which means they can either be untyped values or references to classes. *Primary* permanent relations are general in the sense that they can be used to specify all design patterns. Primary permanent relations can be easily extracted from the structure of the design patterns represented usually by a Unified Modeling Language (UML) [Rambaugh98] class diagram and the collaboration of the pattern participants represented by UML sequence diagrams.

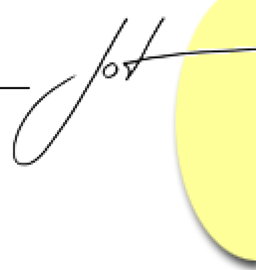


Table 1: Primary Permanent Relations and their Intent

Name	Domain	Intent
Defined-in	***MxC***	Indicates that a method is defined in a certain class.
	AxC	Indicates that an attribute is defined in a certain class.
Reference-to-one (-many)	***CxC***	Indicates that one class defines a member whose type is a reference to one (many) instance(s) of the second class.
Inheritance	***CxC***	Indicates that the first class inherits from the second.
Creation	***MxC***	Indicates that a method contains an instruction that creates a new instance of a class.
	CxC	Indicates that one of the methods of a class contains an instruction that creates a new instance of another class.
Invocation	***MxM***	Indicates that the first method invokes the second method.
Argument	***CxM***	Indicates that a reference to a class is an argument of a method.
	VxM	Indicates that an untyped value is an argument of a method
Instance	***OxC***	Indicates that an object is an instance of a certain class.

Behavioral Aspect

For patterns that have a significant behavioral aspect, it is necessary to understand how objects collaborate to achieve the expected behavior. The subset of TLA used by BPSL to formally specify the behavioral aspect of pattern deals with behaviors $(S_0,S_1,\ldots)$ defined as infinite sequences of states. Each state S_i is a collection of values of state variables (class attributes) and the temporal relations that exist between objects. A pair of consecutive states (S_i,S_{i+1}) in a behavior is called a transition. The system starts in some initial state. As time passes, actions are executed, changing the system's state accordingly. Actions are selected for execution *non-deterministically*, the only restriction being that the precondition of an action must be true in order for the action to be executed. The execution of an action is *atomic,* meaning that once the execution of an action has been started, it cannot be interrupted or interfered with by other actions. The computational model is *interleaving*, that is, only one action at a time is being executed. BPSL has extended the subset of TLA used by extending the semantics of actions. In TLA actions only change state variables (class attributes in pattern terminology). In addition to the above feature, actions in BPSL may also associate and disassociate objects through temporal relations (defined below).

A temporal relation can be defined as follows: *TR(C1[cardinality],C2[cardinality])*, where *TR* is the name of the temporal relation, *C1* and *C2* are classes, and cardinality represents the number of instances of each class that participate in the relation. Cardinality can be represented as either a closed interval *[n..m],* where *n* and *m* represent any two positive integers or *[*]* to depict any possible number of instances.

When used in actions, *TR(o1,o2)* depicts that an object *o1* of a class *C1* is currently associated through *TR* with an object *o2* of a class *C2*, while *¬TR(o1,o2)* depicts that objects *o1* and *o2* are no longer associated through *TR*. *TR(o1,C2)* depicts that object *o1*

is associated with all instances (objects) of the class *C2*. $\neg TR(o1,C2)$ depicts that object *o1* is not associated through *TR* with any object of class *C2*. $\neg TR(C1,C2)$ depicts that none of the objects of class *C1* is associated through *TR* with any object of class *C2*.

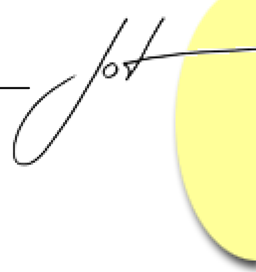
An action consists of a list of parameters (object and untyped values), a precondition and a body. The body is a definition of a state change caused by an execution of the action. For example, if *TR* is a temporal relation between two classes *C1* and *C2*, an action *A* may be defined as follows: $A(o1,o2,p) : TR(o1,o2) \wedge o1.x \neq p \rightarrow \neg TR'(o1,o2) \wedge o1.x'=p$, where *x* is an attribute of the class *C1*, *o1* is an object of the class *C1*, *o2* is an object of the class *C2* and *p* denotes an untyped value. The symbol *":"* means "by definition". Expression $TR(o1,o2) \wedge o1.x \neq p$ is the precondition under which the action can be executed and $\neg TR'(o1,o2) \wedge o1.x'=p$ is the body of the action. The precondition may contain a set of conjunctions and/or disjunctions while the action body may contain a set of conjunctions. Unprimed and primed attributes refer to the values of attributes before and after the execution of the action, respectively. Unprimed and primed temporal relations refer to temporal relations before and after the execution of the action, respectively.

Objects and untyped values that participate in an action are *non-deterministically* selected from those that are suitable. For example, the above action is enabled for all objects having $TR(o1,o2) \wedge o1.x \neq p$. Semantically, an action is a Boolean expression that is true or false with regard to a pair of states. For example the action *A* defined earlier is true for a pair of states *(S,T)* if and only if the value that state *S* assigns to *x* is different from *p* and *o1* is associated with *o2* through *TR* and the value that state *T* assigns to *x* is equal to *p* and *o1* is not longer associated with *o2* through *TR*. Unlike permanent relations which can be used for many patterns, temporal relations and actions are specific to each pattern.

The sequence of states describing the execution of actions is potentially infinite. Properties of a given system (a set of behaviors) can be divided into *safety* and *liveliness*. Informally safety means that nothing will go wrong with the system while liveliness means that some actions will be executed infinitely. Safety can be guaranteed by ensuring that invariants are true at all states of the system while liveliness is obtained by giving an explicit *fairness* requirement. Marking an action with asterisk *(*)* denotes a fairness requirement, stating that if its precondition is true, the action will be executed infinitely often. In BPSL, invariants are defined by the keyword "*Invariants:*", followed by a set of conditions on temporal relations and/or on attribute values. Likewise, initial state is given by the keyword *"Initially:"*, followed by a condition based on temporal relations and/or on attribute values.

Integrating the Structural and Behavioral Aspect Specifications

A formula in BPSL has the following form:



$$\exists (x_1, \ldots, x_n): \begin{cases} \wedge_i \; PR_i \; (a_i,b_i) \;\; \textit{(Permanent relations)} \\ [\wedge_j \; TR_j \; (c_j,d_j) \;\; \textit{(Temporal relations)} \\ \textit{[Invariants: invariant conditions]} \\ \textit{Initially: initial conditions} \\ \vee_k \; A_k \; (\ldots) \;\; \textit{(Actions)]} \end{cases}$$

In the above formula, PR_i are permanent relation symbols, TR_j are temporal relation symbols and A_k are action symbols, while $x_1,..,x_n$ are variable symbols representing the pattern primary entities (classes, attributes, methods, objects and untyped values). In the notation PR_i (a_i,b_i), a_i and b_i could represent classes, attributes, methods, objects or untyped values as per Table 1, while in the notation TR_j (c_j,d_j) c_j and d_j represent classes. The notation A_k *(…)* means that actions can have any number of arguments, which should be either objects or untyped values. Any argument of PR_i, TR_j and A_k must be subset of $\{x_1,..,x_n\}$. Variables $x_1,..,x_n$ are typed, each represents an entity as expected. In all specifications, we follow a convention in which only relations and actions start with a capital letter. Temporal relations, initial conditions, invariants and actions are put between square brackets because they are optional. Patterns that have no significant behavioral aspect are only specified using permanent relations. When patterns have a significant behavioral aspect, initial conditions are compulsory but invariants are optional.

From the parts of a BPSL formula, it can be seen that permanent relations, temporal relations and actions are treated as predicates, i.e., relations whose range is the set of truth values *{true, false}*. This is straightforward for permanent relations. Temporal relations are first defined between classes to indicate that these relations will associate and disassociate objects of these classes when actions execute. As for actions, we have seen previously that they are Boolean expressions (true or false) with regard to a pair of states. This indeed shows that actions are also predicates (with regard to a pair of states). However in addition to their truth or falsity they have additional semantics that derives from TLA. All above explanations justify the usage of the connectives ($\wedge$ and $\vee$) between permanent relations, temporal relations and actions.

BPSL does not use two disjoint subsets of variables, whereby the variables of the first subset participate in permanent relations and the second participate in temporal relations and actions. Indeed, variables $x_1,\ldots,x_n$ participate in permanent and temporal relations as well as actions. This proves that a seamless integration of the two aspects (structural and behavioral) was achieved in BPSL. For example, object variables participate in *Instance* permanent relations while temporal relations, which are heavily used in actions, are defined using class variables. Moreover, in some cases, permanent relations *Reference-to-one(-many)*, *Creation* and *Invocation* can be straightforwardly mapped to temporal relations between objects.

BPSL uses four compartments to specify a pattern. The first declares the variables and their type, the second defines permanent relations, the third defines temporal relations and the fourth defines actions. The $\vee$ connective is used to connect actions

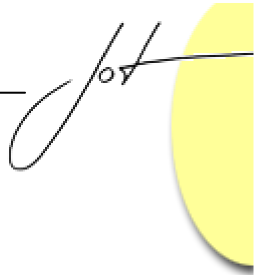
because the action to be executed is non-deterministically chosen from those enabled (their precondition is evaluated to true). This leads to many possible behaviors (infinite sequence of states) as seen previously.

BPSL models a pattern as a collection of entities (classes, attributes, methods, objects, and untyped values) and relations (permanent and/or temporal) and/or actions among them. A *structure* that arises from an instance p of a given pattern shall contain entities that are defined in p and relations (permanent and/or temporal) and/or actions among them. This leads to a *five-sorted* universe of discourse, which is designated by E. The sorts C, A, M, O, V are referred to as *types* or *domains* with respect to variables, relations (permanent and temporal) and actions in BPSL. Therefore, BPSL models patterns using a model M, which is a pair $<E, R>$ where E is a universe of entities, and $R = R_1,\ldots R_n$ is the set of relations (permanent and/or temporal relations and/or actions) amongst. BPSL formulas describe patterns in the form of logic statements. More specifically, every design/program is represented as a *model.* If a BPSL formula φ is the formal specification of a pattern π, a design/program p *conforms* to π if and only if the *model* of p *satisfies* φ.

4 CASE STUDY: THE OBSERVER PATTERN

In this section we use BPSL to specify the *Observer* pattern which is classified in [Gamma95] as a *"behavioral"* pattern. Figure 1 and Figure 2 depict the UML [Rambaugh98] class diagram and the sequence diagram of the *Observer* pattern [Gamma95]. Table 2 depicts the BPSL specification of the *Observer* pattern. It first starts by defining the entities in the system such as classes, attributes, methods, objects and untyped values. This is followed by defining permanent relations between the defined entities. Permanent relations specify the structural aspect of the pattern. All permanent relations defined in this specification are *primary* as shown in Table 1.

The permanent relation *Defined-In* depicts which attribute or method belongs to which class.The permanent relations *Reference-to-one* and *Reference-to-many* depict, in this case, that a *concrete-observer* has only one reference to a *concrete-subject* while a *subject* has many references to an *observer*. *Inheritanc*e depicts the classical relationship between a base and a derived class. All the above relations are straightforwardly derived from the class diagram of Figure 1. The *Invocation* relations *Invocation(set-state, notify*), *Invocation(notify, update)* and *Invocation(update, get-state)* are straightforwardly derived from the sequence diagram of Figure 2. The relation *Argument* depicts that a reference to a class is a parameter of a given method. In the case of our specification we have: *Argument(observer, attach), Argument(observer, detach)* and *Argument(subject, update)*. The permanent relations *Instance(s, concrete-subject)* and *Instance(o, concrete-observer)* depict that *s* is an object of the class *concrete-subject* while *o* is an object of the class *concrete-observer*. These objects will be later used in the specification of actions.



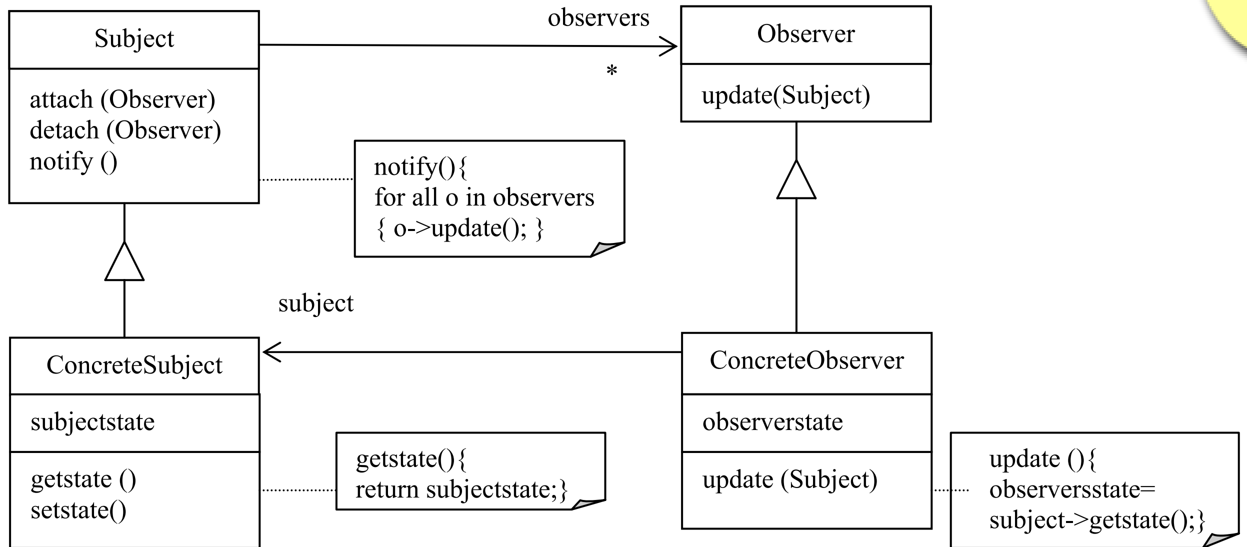


Fig.1: UML class diagram of the *observer* pattern

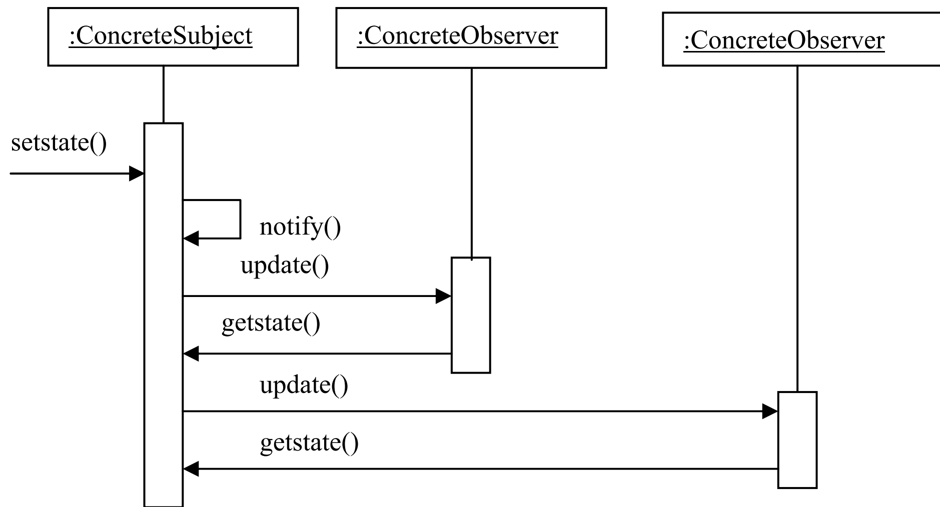
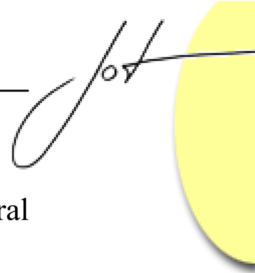


Fig.2: UML sequence diagram of the *observer* pattern

Table 2: BPSL specification of the *observer* pattern

$\exists$ *subject, concrete-subject, observer, concrete-observer* $\in C$; *subject-state, observer-state* $\in A$; *attach, detach, notify, get-state, set-state, update* $\in M$; *o, s* $\in O$; *d* $\in V$;
Defined-in(subject-state, concrete-subject) $\wedge$ *Defined-in (observer-state, concrete-observer)* $\wedge$ *Defined-in(attach, subject)* $\wedge$ *Defined-in (detach ,subject)* $\wedge$ *Defined-in (notify, subject)* $\wedge$ *Defined-in(set-state, concrete-subject)* $\wedge$ *Defined-in (get-state, concrete-subject)* $\wedge$ *Defined-in (update, observer)* $\wedge$ *Reference-to-one(concrete-observer, concrete-subject)* $\wedge$ *Reference-to-many(subject, observer)* $\wedge$ *Inheritance(concrete-subject, subject)* $\wedge$ *Inheritance(concrete-observer, observer)* $\wedge$ *Invocation(set-state, notify)* $\wedge$ *Invocation(notify, update)* $\wedge$ *Invocation(update, get-state)* $\wedge$ *Argument(observer, attach)* $\wedge$ *Argument(observer, detach)* $\wedge$ *Argument(subject, update)* $\wedge$ *Instance(s, concrete-subject)* $\wedge$ *Instance(o, concrete-observer).*
Attached(concrete-subject[0..1],concrete-observer[*]) $\wedge$ *Updated(concrete-subject[0..1],concrete-observer[*]).*
Initially: $\neg$*Attached(s, concrete-observer).* *Attach(s,o) :* $\neg$*Attached(s,o)* $\rightarrow$ *Attached'(s,o)* $\vee$ *Detach(s,o): Attached(s,o)* $\vee$ *Updated(s,o)* $\rightarrow$ $\neg$*Attached'(s,o)* $\vee$ *Notify(s,o,d) : Attached(s,o)* $\vee$ *Updated(s,o)* $\rightarrow$ $\neg$*Updated'(s,concrete-observer)* $\wedge$ *s.subject-state' =d* $\vee$ *Update*(s,o):* $\neg$*Updated(s,o)* $\rightarrow$ *Updated'(s,o)* $\wedge$ *o.observer-state' = s.subject-state.*

In the third and fourth part of the pattern specification, temporal relations and actions are defined. These specify the behavioral aspect of the pattern. In the *Observer* pattern, an instance of a *concrete-observer* is associated with an instance of a *concrete-subject* whenever it is interested in its content. This is shown by the following temporal relation: *Attached(concrete-subject[0..1],concrete-observer[*])*. The cardinality shows that many *concrete-observers* might be attached to zero or one *concrete-subject*. Moreover, each *concrete-subject* needs to know to which *concrete-observers* its contents have been delivered since the last modification. Thus *concrete-subjects* are associated with



concrete-observers that have already been updated. This yields the following temporal relation: *Updated (concrete-subject[0..1], concrete-observer[*])*.

The initial condition *"¬Attached(s, concrete-observer)"* reflects that initially all *concrete-observers* are not attached to the *concrete-subject (s)*.In this pattern a *concrete-observer* can become interested in the contents of a *concrete-subject*, and may also cancel this interest. In the specification actions *Attach* and *Detach* are used for modeling these cases respectively. Action *Attach* sets *Attached* relation between the *concrete-subject* and the *concrete-observer* objects that are involved. Action *Detach* clears this relation. These actions are formalized as follows: *Attach(s,o) :¬Attached(s,o) → Attached'(s,o)* and *Detach(s,o): Attached(s,o) ∨ Updated(s,o) → ¬Attached'(s,o)*. The precondition of the action *Detach* contains a disjunction because *s* and *o* could either be linked by the temporal relation *Attached* or *Updated* when the action *Detach* takes place. Action *Notify* denotes that the contents of a *concrete-subject* have been modified. This requires updating all *concrete-observers* that are attached to this *concrete-subject*. The precondition of *Notify* is the same as *Attach*, object *s* and *o* should either be linked by *Attached* or *Updated*. Thus, upon executing *Notify*, the *concrete-subject* must no longer be associated with any *concrete-observer* by *Updated* relation. This yields the following action: *Notify(s,o,d) : Attached(s,o) ∨ Updated(s,o) → ¬Updated'(s,concrete-observer) ∧ s.subject-state' =d* where *d* denotes the new value, set upon notification. The usage of *concrete-observer* in the temporal relation *¬Updated'(s,concrete-observer)* denotes that all instances of the class *concrete-observer* will become not updated. As no restrictions are imposed on the value of parameter d, its value is non-deterministically selected. Action *Update* represents a transmission of modified data from a *concrete-subject* to a *concrete-observer*. Thus, it sets *Updated* relation for the *concrete-subject* and the *concrete-observer* that participate in the action, yielding: *Update*(s,o): ¬Updated(s,o) → Uptated'(s,o) ∧ o.observer-state' = s.subject-state*. Marking the action with an asterisk *(*)* denotes a *fairness* requirement, stating that if its precondition is true, the action will be executed infinitely often.

5 RELATED WORK

Object Constraint Language (OCL) [Warmer98] was developed by IBM to accurately specify constraints that cannot be unambiguously described by UML graphical models. It is a formal language that is intended to be easy to read and write. OCL is a pure expression language. All values for all objects, including all links, will not change. Whenever an OCL expression is evaluated, it simply delivers a value. OCL can be used for a number of different purposes:

- To specify invariants on classes and types in the class model.
- To specify type invariant for Stereotypes.
- To describe pre- and post conditions on Operations and Methods.
- To describe Guards.

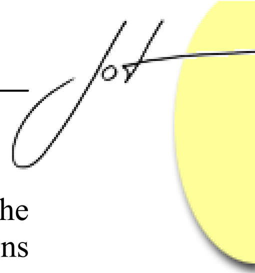
- As a navigation language.
- To specify constraints on operations.

OCL is used in the *UML Semantics* document to specify the well-formedness rules of the UML meta-model. Each well-formedness rule in the static semantics sections in the *UML Semantics* document contains an OCL expression, which is an invariant for the involved class. BPSL and OCL have very little in common. OCL is an add-on to better understand UML graphical models while BPSL is a language that is meant to accurately describe design patterns. Since design patterns are described using textual descriptions, UML graphical models and sample code fragments, someone could think of using OCL to specify constraints of the UML graphical models of the design pattern. However, this will only help better understand those graphical models rather than better understanding the essence of the design pattern.

Contracts are descriptions of obligations among clients who utilize the services provided by suppliers. Design by Contract is a technique where pre-conditions, post-conditions, and invariants are used to define contracts [Helm90]. Pre-conditions are Boolean assertions that a client must satisfy before requesting a service. Post-conditions are Boolean assertions that a supplier must satisfy after providing a service. Invariants are Boolean assertions that must be satisfied over time. Actions in BPSL are quite similar to contracts in the sense that they define precondition as well as show the state change. However in BPSL state change is not only reflected by change in attribute values but also by associating and disassociating objects through temporal relations.

The Abstract Data View (ADV) approach [Cowan95], uses a formal model to achieve separation by dividing designs into two types of components: objects and object views and by strictly following a set of design rules. Specific instantiations of views as represented by Abstract Data Views (ADVs) and objects called Abstract Data Objects are substituted into the design pattern realization while maintaining a clear separation between view and object. ADV and ADO components are specified using temporal logic and the interconnection between components is described in terms of category theory. Clearly, ADV/ADO concentrate on formalizing the process of instantiating a solution in given programming language from a design pattern while BPSL seeks a multi-purpose formal specification language as highlighted in the introduction.

Our work is mostly inspired by LanguagE for Patterns' Uniform Specification (LePUS) [Eden99] and Distributed Co-operation (DisCo) [Mikkonen98]. LePUS derives from Higher-Order logic and focuses only on specifying the structural aspects of design patterns. We preferred to use a small fraction of FOL because simplicity was paramount in the design of BPSL. If the users of a formal specification language for design patterns cannot easily understand it, how are they supposed to understand design patterns formally specified by this language? DisCo derived from TLA and was designed to specify reactive systems, which are in constant interaction with their environment and therefore have a predominantly behavioral aspect. DisCo has little (almost no) support for specifying the structural aspect. The subset of TLA used in BPSL is different from the one used in DisCo, the syntax is completely different while they share most of the



semantics derived from TLA concepts. DisCo and in fact TLA itself does not support the concept of temporal relations and its semantics as defined in section 3. In DisCo actions change only state variables while in BPSL they change state variables (class attributes) as well as associate and disassociate objects through temporal relations.

6 CONCLUSION

Formal specification of design patterns is intended to complement existing textual and graphical descriptions in order to eliminate ambiguity, allow rigorous reasoning about patterns and facilitate automation of the activities related to them. As patterns represent abstractions, any formal language meant to specify them should strive to achieve simplicity for a better understandability, accuracy for a precise semantics and completeness to avoid loss of semantics.

Since patterns have two complementary aspects (structural and behavioral), BPSL was devised to combine the specification of the two aspects in order to achieve completeness. BPSL has carefully chosen the subsets of FOL and TLA to be used in order for it to be simple for users and yet describe design patterns accurately. The ultimate purpose of BPLS is to help users understand patterns to know exactly when and how to use them.

The two classical schools of thought (structural vs. behavioral) that originated from modeling OO systems have unfortunately been inherited by the pattern specification community. The structural school of thought claims that the structural aspect of design patterns is predominant in all systems and even that the behavioral aspect can be derived from it [Eden01]. The behavioral school of thought concentrated on specifying patterns for reactive systems, which have a predominantly behavioral aspect. In BPSL a design pattern encompasses both views (structural and behavioral) in a complementary manner.

RFERENCES

[Chinn99] Chinnasamy, S., Raje, R.R., and Liu, Z., "Specification of design patterns: An analysis", *Proceedings of the 7th International Conference on Advanced Computing and Communications* (ADCOM'99), pp. 300-304, 1999.

[Cowan95] Cowan, D.D., and Lucena, C.J. P. "Abstract data views: An interface specification concept to enhance design for reuse", *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 229-243, 1995.

[Eden99] Eden , A. H., Hirshfeld, Y., and Lundqvist, K., "LePUS–Symbolic logic modeling of object oriented architectures: A case study", *Second Nordic Workshop on Software Architecture* (NOSA'99), 1999.

[Eden01] Eden, A.H., and Hirshfeld, Y., "Principles in formal specification of object-oriented architectures", *CASCON'01*, 2001.

[Gamma95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design patterns: Elements of reusable object-oriented systems*, Addison-Wesley, 1995.

[Helm90] Helm, R., Holland, I.M., and Gangopadhyay, D., "Contracts: Specifying behavioral compositions in object-oriented systems", *Proceedings of ECOOP/OOPSLA'90*, pp. 169-180, 1990.

[Lampo94] Lamport, L., "The temporal logic of actions", *ACM Transactions on Programming Languages and Systems*, vol.16, no. 3, pp. 872-923, 1994.

[Mikko98] Mikkonen, T., "Formalizing design patterns", *Proceedings of ICSE'98*, pp. 115-124, 1998.

[Ramba98] Rambaugh, J., Jacobson, I., and Booch, G., *The unified modeling language reference manual*, Addison-Wesley, 1998.

[Taibi01] Taibi, T., and Ngo, D.C.L, "Why and how should patterns be formalized", *Journal of Object-Oriented Programming (JOOP)*, vol. 14, no. 4, pp. 8-9, 2001.

[Vliss97a] Vlissides , J. M., "Multicast", *C++ Report*, Sep. 1997.

[Vliss97b] Vlissides, J. M., "Multicast - Observer = Typed Message", *C++ Report*, Nov.-Dec. 1997.

[Warme98] Warmer J., and Kleppe, A. G., *The object constraint language precise modeling with UML*, Addison-Wesley, 1998.

About the authors

Toufik Taibi is a lecturer at the Faculty of Information Technology, Multimedia University, Cyberjaya, Malaysia. His research interests include formal specification of design patterns, distributed object computing, object-oriented methods and software engineering. He can be reached at toufik.taibi@mmu.edu.my.

Dr. David Chek Ling Ngo is associate professor and Dean of the Faculty of Information Science and Technology at Multimedia University, Melaka, Malaysia. His current research interests center on arithmetic aesthetics, proportional and design systems, and screen design. He can be reached at david.ngo@mmu.edu.my.