Automatic Proof of Refinement among Design Patterns using the TLC Model Checker

TOUFIK TAIBI
College of Information Technology
United Arab Emirates University
P.O. Box 17555 Al Ain
UNITED ARAB EMIRATES
e-mail: toufikt@uaeu.ac.ae

ÁNGEL HERRANZ
Facultad de Informática,
Universidad Politécnica de Madrid,
Campus de Montegancedo s/n Boadilla
Del Monte, 28660 Madrid, SPAIN
e-mail:aherranz@fi.upm.es

Abstract: - Design patterns are reuse artifacts meant to improve the quality of software designs as well as the productivity of designers. Patterns (and their relationships) are mostly described in an informal fashion which leads to ambiguity and limits tools support. This has worsened with the growing number of well-established and candidate patterns. This paper discusses how to formally specify the "solution element" of patterns and their relationships using TLA+, the formal specification language of Temporal Logic of Actions (TLA). The paper first classifies and formally defines the most common relationships between patterns. Then, it shows how to automatically proof the existence of a *"refines"* relationship between patterns using TLC– the TLA+ Model Checker.

Keywords: - Temporal Logic of Action (TLA), TLA+, temporal relations, actions, refinement, TLC.

1. Introduction

Design patterns represent the culmination of many years in which experienced designers were solving problems repeatedly encountered within certain contexts. Hence, reusing patterns yields better quality software within reduced time frame. Patterns are usually described in catalogs. The last decade has seen the publication of many such catalogs [3][15][8][4][12]. Most pattern writers use a combination of textual descriptions, Object-Oriented (OO) graphical notations [10] and sample code fragments to describe patterns. Informal specifications are ambiguous and sometimes misleading in understanding and properly applying patterns. Hence, there is a need for a formal means to accurately describe patterns.

As the number of patterns (well established and candidate patterns) is growing, it is of major importance that relationships between patterns are described precisely in order to facilitate the correct usage of patterns. Unfortunately, pattern catalogs do not describe these relationships in a consistent manner but rather each uses its own classification terminology. This paper discusses how to formally specify patterns and their relationships using TLA+ [7], the formal specification language of Temporal Logic of Actions (TLA) [6]. This works builds-up from the work done in [11] on Balanced Pattern Specification Language (BPSL).

This paper first classifies and formally defines the most common relationships between patterns. Then, it shows how to automatically proof the existence of a *"refines"* relationship between patterns using TLC [7]– the TLA+ Model Checker.

This work as many others in this field focuses on formally specifying the solution element of a pattern and not on its other elements such as the problem solved, the context, the important forces [1] acting within the problem, consequences, etc. The reason being that the verbal description of the solution element is the most coherent and the most tangible to formalize.

The rest of the paper is organized as follows. Section 2 provides a detailed description of how patterns are formally specified using TLA+. Section 3 classifies the relationships between patterns and formally describes them. Section 4 provides a case study, while section 5 concludes the paper.

2. Formal Specification of Design Patterns Using TLA+

TLA is a logic for specifying and reasoning about concurrent and reactive systems. A typical TLA formula has the form: $Init \wedge \Box[Next]_u \wedge Liveness$. *Init* is the initial-state predicate—a formula describing all legal initial states. $[Next]_u$ is the next-state relation, which specifies all possible steps (pairs of successive states) in a behavior of the system. The subscript u is a tuple of flexible variables and the notation $[Next]_u$ allows *stuttering* steps in which variables in u do not change. *Next* is a disjunction of actions that describe the different system operations. An action is a mathematical formula in which

unprimed variables refer to the first state of a step and primed variables refer to its second state. Actions can contain parameters symbols which do not represent known values like *1* or *"abc"*. However, unlike *flexible variables*, the value of a parameter does not change. It must be the same in the old and new state. The parameter denotes some fixed but unknown value. It is thus called a *rigid variable*. *Liveness* is a temporal formula that specifies the liveness (progress) properties of the system as the conjunction of fairness conditions (usually *Weak Fairness* denoted *WF* in the case of pattern specification) on actions.

The structural aspect of patterns is represented by sub-classes participating in the pattern and the association between them. Classes are represented as sets of instances (objects), each of which is represented by an identity taken from an infinite set of object identities. As such we use the terms object and object identity interchangeably.

The behavioral aspect of patterns is described as set of behaviors. New states are produced through the execution of actions. Each state is defined by the values of temporal relations. Temporal relations are mathematical relations defined between objects of two classes. A temporal relation is thus a subset of the Cartesian product of the two sets (classes). Temporal relations are defined as TLA flexible variables. They have been called "temporal" because their value changes over time while actions are being executed.

Associations between sub-classes participating in the pattern generate the "main" temporal relations while the other temporal relations are derived from the "main" ones. For example in the specification of MEDIATOR_1, the temporal relation *Connected* is generated from the association between the classes *concrete_mediator* and *concrete_colleague*, while the temporal relations *Sent* and *Called* are "derived" from *Connected*.

Temporal relations are means of providing an abstract way of specification such that low-level programming details are avoided. In later low-level versions of the specification, temporal relations can be defined as implementation-level TLA variables.

The structure of a TLA+ formula for specifying patterns is shown in Table 1. The theorems reflect that the execution of the actions preserve invariants (which at the minimum contain type definitions of flexible variables) and satisfy pattern properties.

Table 1, Structure of a TLA+ Formula for Specifying Patterns

$Invariants \triangleq I_1 \wedge ... \wedge I_k$	*{Pattern invariants}*
$Properties \triangleq P_1 \wedge ... \wedge P_l$	*{Pattern Properties}*
$Init \triangleq P$	*{P is the initial predicate}*
$Next \triangleq A_1 \vee ... \vee A_m$	*{A_1...A_m are m actions each of which could have rigid variables}*
$u=\langle u_1,...,u_n \rangle$	*{tuple of n flexible variables}*
$Spec \triangleq Init \wedge \Box[Next]_u \wedge WF_u(A)$	*{$A=A_{i1} \vee ... \vee A_{i2}, 1<=i_1<=i_2<=m$ }*
$Theorem\ Spec \Rightarrow \Box Invariants$	*{Ensuring pattern invariants are always preserved}*
$Theorem\ Spec \Rightarrow Properties$	*{Ensuring pattern properties are satisfied}*

3. Classification and Formal Specification of Patterns Relationships

There are not many articles investigating the relationships between patterns. Nonetheless, the most prominent studies were done by Nobel [9] and Zimmer [16]. Our work builds on Nobel's classification of primary relationships. However, we have provided our own version of the definitions of these relationships in addition to providing the mathematical definition for them. Relationships between patterns can be classified in three categories: *"uses"*, *"refines"*, *"differs from"* and *"equivalent to"*. The next sections describe each of these relationships.

3.1. "Uses"

The *"uses"* relationship is the most common form of pattern relationships. Informally it describes that a *"bigger"* pattern is made of *"smaller"* patterns. For example, the MODEL-VIEW-CONTROLLER pattern [2] can be seen as a composite of the OBSERVER, STRATEGY, and COMPOSITE patterns [5]. Formally, the *"uses"* relationship can be defined as follows. If P is a pattern made-of patterns P_1,..,P_n each specified using formulas Ψ_1,…, Ψ_n respectively and if Q is a pattern specified using formula Φ, *P "uses" Q* if and only if (iff): $\exists i : \Psi_i \Leftrightarrow \Phi$.

3.2. "Refines"

A pattern *"refines"* another pattern, if one pattern is a specialization of a more general, simpler, or more abstract pattern. The "*uses"* relationship is similar to composition, while the *"refines"* relationship is similar to inheritance. For example, TYPED MESSAGE [14] is a refinement of MULTICAST [13]. Formally, the *"refines"* relationship can be defined as follows. If P is a pattern specified using formula Ψ and if Q is a pattern specified using formula Φ, *Q "refines" P* iff: $\Phi \Rightarrow \Psi$. Note that the above is done with proper substitutions of flexible variables.

3.3. "Differs From" and "Is Equivalent to"

A pattern *"differs from"* another pattern if they provide mutually exclusive solutions to their

problems (their solutions have nothing in common). Formally, the *"differs from"* relationship can be defined as follows. If *P* is a pattern formalized using formula Ψ and if *Q* is pattern formalized using formula Φ, *P "differs from" Q* iff: $\neg(\Psi \Rightarrow \Phi) \wedge \neg(\Phi \Rightarrow \Psi)$. The antonym of the above relationship is *"equivalent to"*. If *P* is a pattern formalized using formula Ψ and if *Q* is a pattern formalized using formula Φ, *P "is equivalent to" Q* iff: $(\Psi \Phi)$.

4. Case Study

This section describes two versions of the MEDIATOR pattern [5] which we call MEDIATOR_1 and MEDIATOR_2. We will prove using TLC that the two patterns have equivalent specifications.

4.1 MEDIATOR_1 Pattern

Figure 1 depicts the structure of the most abstract form of *MEDIATOR_1* pattern. *Connected*, *Sent* and *Called* are temporal relations defined between *concrete_mediator* and *concrete_colleague*. "*" represent the cardinality of the relations which is many-to-many.

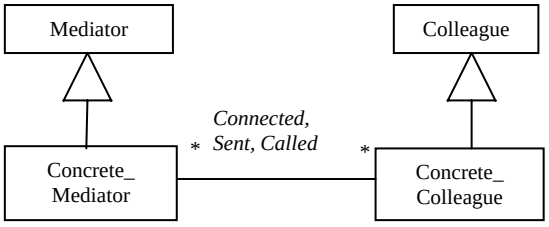

Figure 1, Structure of *MEDIATOR_1* Pattern

The *MEDIATOR_1* pattern has the following behavioral elements. *A concrete_colleague 'c'* can connect to a *concrete_mediator 'm'* showing that it wants to communicate (through it) its data change to other *concrete_colleagues*. This is reflected by action *Connect(m,c)*. A data change occurs in a connected *concrete_colleague 'c'* and it informs a *concrete_mediator 'm'* about it. This is reflected by action *Send (m,c,G)*, in which 'G' is the set of *concrete_colleagues* which will be called by *'m'*. *'G'* is defined by *'m'* and not *'c'*. The above change triggers that a *concrete_mediator* sends messages to other *concrete_colleagues* based on its internal knowledge. This is reflected by action *Call* in which, *'m'* is a *concrete_mediator*, *'c'* is the *concrete_colleague* which sent the message to *'m'* and *'G'* is the set of *concrete_colleagues* to be called by *'m'*. A connected *concrete_colleague 'c'* can disconnect from a *concrete_mediator 'm'* showing it no longer wants to communicate (through it) its data change to other *concrete_colleagues*. This is reflected by action *Disconnect(m,c)*.

Table 2, TLA+ Specification of MEDIATOR_1 Pattern

```
---------------------- MODULE Mediator_1 ----------------------
CONSTANT concrete_mediator, concrete_colleague
VARIABLE Connected, Sent, Called
Inv_1 == /\ (Connected \subseteq concrete_mediator \X
             concrete_colleague)
         /\ (Sent \subseteq Connected)
         /\ (Called \subseteq Connected)
Init_1== /\ Connected={}
         /\ Sent={}
         /\ Called={}
Connect(m,c) == /\ <<m,c>> \notin Connected
          /\ Connected'=Connected \union {<<m,c>>}
          /\ Called'=Called \union {<<m,c>>}
          /\ UNCHANGED Sent
Send(m,c,G)== /\ <<m,c>> \in Connected
         /\ <<m,c>> \notin Sent
         /\ G # {}
         /\ G \in SUBSET concrete_colleague
         /\ c \notin G
         /\ \A y \in G: <<m,y>> \in Connected
         /\ {<<x,y>> \in (concrete_mediator \X G) : <<x,y>> \in
            Called /\ x=m}={}
         /\ Sent'=Sent \union {<<m,c>>}
         /\ Called'=Called \ {<<x,y>> \in (concrete_mediator \X
            G):x=m}
         /\ UNCHANGED Connected
Call== \E m \in concrete_mediator, c \in concrete_colleague, G \in
         SUBSET concrete_colleague :
         /\ <<m,c>> \in Connected
         /\ <<m,c>> \in Sent
         /\ G # {}
         /\ c \notin G
         /\ (\A y \in G: <<m,y>> \in Connected)
         /\ (\A y \in G: <<m,y>> \notin Called)
         /\ Called'=Called \union {<<x,y>> \in (concrete_mediator
            \X  G):x=m}
         /\ Sent'=Sent\{<<m,c>>}
         /\ UNCHANGED Connected
Disconnect(m,c)== /\ <<m,c>> \in (Connected \intersect Called)
                  /\ Connected'=Connected\{<<m,c>>}
                  /\ Called'=Called\{<<m,c>>}
                  /\ Sent'=Sent\{<<m,c>>}
Next_1 ==(\E m \in concrete_mediator, c \in concrete_colleague:
           Connect(m,c)) \/
          (\E m \in concrete_mediator, c \in concrete_colleague,
          G \in SUBSET concrete_colleague: Send(m,c,G)) \/
          Call \/
          (\E m \in concrete_mediator, c \in
          concrete_colleague:Disconnect(m,c))
v1 == <<Connected,Sent, Called>>
Spec_1==  Init_1 /\ [][Next_1]_v1 /\ WF_v1 (Call)
THEOREM  Spec_1 => []Inv_1
=====================================
```

Table 2 depicts the TLA+ specification of the MEDIATOR_1 pattern. TLA+ is well described in [7] and due to space limitations, we do not intent to further detail it here. Table 3 depicts the file *Mediator_1_Refines_Mediator_2.tla*, which extends the *Mediator_1* module and creates an instance of *Mediator_2* module with the proper substitutions of constants and variables defined in both patterns. Moreover, a theorem was defined to show the specification of the MEDIATOR _1 pattern (*Spec_1*)

implies the one of the MEDIATOR_2 (*Spec*_2 defined as *Med_2!Spec_2*).

Table 3, File Mediator_1_Refines_Mediator_2.tla

```
----------------------MODULE Mediator_1_Refines_Mediator_2-------
EXTENDS Mediator_1
Med_2==INSTANCE Mediator_2 WITH
    _concrete_mediator <- concrete_mediator,
    subject_colleague <- concrete_colleague,
    observer_colleague <- concrete_colleague,
    _Connected<-Connected,
    _Sent<-Sent,
    _Called<-Called
Spec_2==Med_2!Spec_2
THEOREM Spec_1=>Spec_2
=============================================
```

4.2 MEDIATOR_2 Pattern

Figure 2 depicts the structure of the most abstract form of *MEDIATOR_2* pattern. In addition to the behavioral elements found in MEDIATOR_1, MEDIATOR_2 has the following additional features:

- The pattern has two *concrete_colleagues* which are *subject_colleague* and *observer_colleague.*
- Only *subject_colleagues* are allowed to send messages through the *concrete_mediator* while only *observer_colleagues* are allowed to receive them.

Indeed, this version of the MEDIATOR pattern introduces participants of the OBSERVER pattern [5], in which *concrete_subjects* do not communicate directly with *concrete_observers* but do that through a *concrete_mediator*.

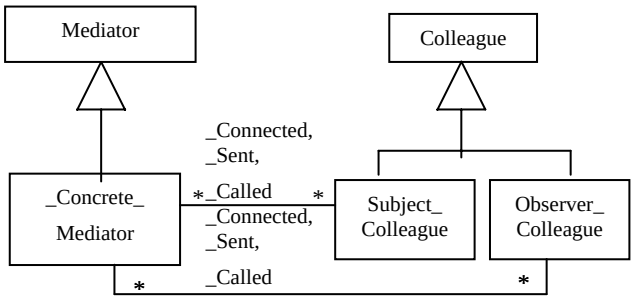

Figure 2, Structure of *MEDIATOR_2* Pattern

Table 4 depicts the TLA+ specification of MEDIATOR_2, in which, changes in the specification as compared to Table 2 have been highlighted in bold. Name substitutions have been achieved by adding *"_"* to the names of classes, temporal relations and actions. Table 5 depicts the file *Mediator_2_Refines_Mediator_1.tla*, which extends the *Mediator_2* module and creates an instance of *Mediator_1* module with the proper substitutions of constants and variables defined in both patterns. Moreover, a theorem was defined to show the specification of the MEDIATOR _2 pattern (*Spec_2*) implies the one of the MEDIATOR_1 (*Spec*_1 defined as *Med_1!Spec_1*).

Table 4, TLA+ Specification of MEDIATOR_2 Pattern

```
---------------------- MODULE Mediator_2 ----------------------
CONSTANT subject_colleague, observer_colleague,
          _concrete_mediator
VARIABLE _Connected, _Sent, _Called
Inv_2== /\ (_Connected \subseteq _concrete_mediator \X
          (subject_colleague \union observer_colleague))
         /\ (_Sent \subseteq _Connected)
         /\ (_Called \subseteq _Connected)
Init_2== /\ _Connected={}
         /\ _Sent={}
         /\ _Called={}
_Connect(m,c) ==  /\ <<m,c>> \notin _Connected
                  /\ _Connected'=_Connected \union {<<m,c>>}
                  /\ _Called'= IF c \in observer_colleague THEN
                     _Called \union {<<m,c>>} ELSE _Called
                  /\ UNCHANGED _Sent
_Send(m,c,G)==  /\ c \in subject_colleague
                /\ <<m,c>> \in _Connected
                /\ <<m,c>> \notin _Sent
                /\ G # {}
                /\ G \in SUBSET observer_colleague
                /\ \A y \in G: <<m,y>> \in _Connected
                /\ {<<x,y>> \in (_concrete_mediator \X G) :
                   <<x,y>> \in _Called /\ x=m}={}
                /\ _Sent'=_Sent \union {<<m,c>>}
                /\ _Called'=_Called \ {<<x,y>> \in
                  (_concrete_mediator \X G):x=m}
                /\ UNCHANGED _Connected
_Call== \E m \in _concrete_mediator, c \in subject_colleague, G
          \in  SUBSET observer_colleague :
         /\ <<m,c>> \in _Connected
         /\ <<m,c>> \in _Sent
         /\ (\A y \in G: <<m,y>> \in _Connected)
         /\ _Called'=_Called \union {<<x,y>> \in
            (_concrete_mediator \X G):x=m}
         /\ _Sent'=_Sent\{<<m,c>>}
         /\ UNCHANGED _Connected
_Disconnect(m,c)==  /\ <<m,c>> \in (_Connected \intersect
                      _Called)
                    /\ _Connected'=_Connected\{<<m,c>>}
                    /\ _Called'= IF c \in observer_colleague
                      THEN _Called \ {<<m,c>>} ELSE _Called
                    /\ _Sent'=_Sent\{<<m,c>>}
Next_2 ==(\E m \in _concrete_mediator, c \in (subject_colleague
          \union observer_colleague):  _Connect(m,c)) \/
          (\E m \in _concrete_mediator, c \in subject_colleague, G
          \in SUBSET observer_colleague: _Send(m,c,G)) \/
          _Call \/
          (\E m \in _concrete_mediator, c \in (subject_colleague
          \union observer_colleague):_Disconnect(m,c))
v2 == <<_Connected,_Sent,_Called>>
Spec_2 ==  Init_2 /\ [][Next_2]_v2 /\ WF_v2 (_Call)
THEOREM Spec_2 =>[]Inv_2
=============================================
```

Table 5, File Mediator_1_Refines_Mediator_2.tla

```
----------------------MODULE Mediator_2_Refines_Mediator_1-------
EXTENDS Mediator_2
Med_1==INSTANCE Mediator_1 WITH
    concrete_mediator <- _concrete_mediator,
    concrete_colleague<- subject_colleague \union observer_colleague,
    Connected<-_Connected,
    Sent<-_Sent,
    Called<-_Called \union {<<x,y>> \in _concrete_mediator \X
subject_colleague: <<x,y>> \in _Connected}
```

```
Spec_1==Med_1!Spec_1
THEOREM Spec_2=>Spec_1
=======================================
```

4.3 Refinements Proof Using TLC Model Checker

TLA+ models can be validated and verified in order to make sure that a model faithfully reflects the intended system. Model checkers can explore traces allowed by the model, possibly detecting deadlock or violation of invariants. Moreover, they can assist in the formal verification of properties. TLC, the TLA+ model checker can be used for verification and validation of specifications written in TLA+. TLC can analyze the state space of finite instances of TLA+ models. In addition to the TLA+ model written in an ASCII representation (like the ones in Table 2-Table 5), TLC requires a configuration file that defines the finite-state instance to analyze and declares the specifications and the properties to verify. TLC needs to know explicitly (thorough the configuration file) which of the formulas represent the system specification to analyze and which theorem needs to be interpreted. Table 6 shows the configuration files of the TLA+ files Mediator_1_Refines_Mediator_2.tla and Mediator_2_Refines_Mediator_1.tla.

Table 6, TLC Configuration files of Mediator_1_Refines_Mediator_2.tla and Mediator_2_Refines_Mediator_1.tla

MEDIATOR_1 Pattern	MEDIATOR_2 Pattern
SPECIFICATION Spec_1 *INVARIANT Inv_1* *PROPERTY Spec_2* *CONSTANTS* *concrete_mediator={m1,m2}* *concrete_colleague={c1,c2,c3,c4}*	*SPECIFICATION Spec_2* *INVARIANT Inv_2* *PROPERTY Spec_1* *CONSTANTS* *_concrete_mediator={_m1,_m2}* *subject_colleague={s1,s2}* *observer_colleague={o1,o2}*

The configuration files define concrete instances of TLA+ modules Mediator_1_Refines_Mediator_2 and Mediator_2_Refines_Mediator_1 by defining the sets *concrete_mediator*, *concrete_colleague*, *_concrete_mediator*, *subject_colleague* and *observer_colleague*. The keyword *SPECIFICATION* indicates the formula representing the main system specification. Properties to be checked are specified with the *PROPERTY* statement. This means that TLC checks if *Spec* $\Rightarrow$ *Prop* is valid for the entire state space. Invariants to be checked are specified with the statement *Spec* $\Rightarrow \Box$*Inv* is valid which requires checking that *Spec* $\Rightarrow \Box$*Inv* for every step of a behavior.

Figure 3 and 4 show windows in which TLC was run on both specifications (Mediator_1_Refines_Mediator_2 and Mediator_2_Refines_Mediator_1). Both were correct and showing indeed that MEDIATOR_1 is a refinement of MEDIATOR_2 (and vice-versa). As such the two specifications are indeed equivalent.

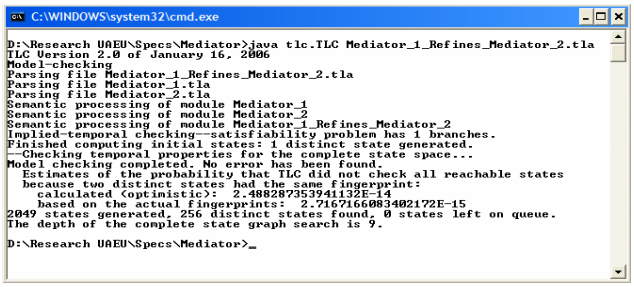

Figure 3, Running TLC on the Mediator_1_Refines_Mediator_2 Specification

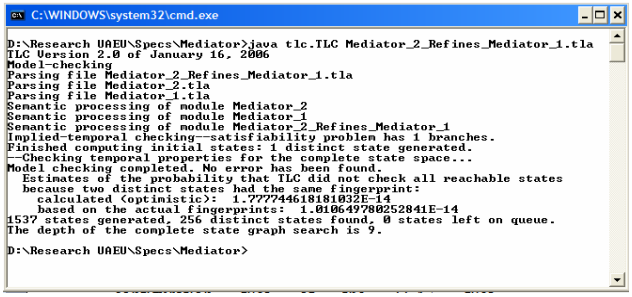

Figure 4, Running TLC on the Mediator_2_Refines_Mediator_1 Specification

TLC firsts checks the syntactic and semantic correctness of a TLA+ specification. It then computes the graph of reachable states for the instance of the model defined by the configuration file, while verifying the invariants. Finally, the temporal properties are verified over the state space. Trying to analyze somewhat larger models, leads to the well-known problem of state-space explosion.

5. Conclusion

The inherent benefits of patterns cannot be fully exploited by the existing informal means of specification. Formal specification of patterns brings accuracy and facilitates tool support. This allows rigorous reasoning about patterns and their relationships. This intent was shown in this paper using two versions of the *MEDIATOR* patterns as a case study. Using TLA+, we were able to formally specify the most abstract form of these patterns without dealing with implementation details in contrast to other pattern formalization approaches. Moreover, using TLC we were able to check the correctness of both specifications and prove that the

specifications of MEDIATOR_1 and MEDIATOR_2 patterns are equivalent.

References:

[1] Alexander, C., *A Pattern Language*. Oxford University Press, 1977.

[2] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal. M. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[3] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1996.

[4] N. Harrison, B. Foote and H. Rohnert (Editors), *Pattern Languages of Program Design*, Volume 4, Addison-Wesley, 1999.

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

[6] Lamport, L. The temporal logic of actions. *ACM transactions on Programming Languages and Systems*, 16, 3 (1994). 872-923.

[7] Lamport L., Specifying Systems: *The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Professional, 2002.

[8] Martin, R.C., Riehle, D., and Buschmann, F. (Editors), *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1998.

[9] Noble, J., Classifying Relationships between Object-Oriented Design Patterns. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, Adelaide, IEEE Computer Society Press pp. 98-107, 1998.

[10] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[11] Taibi, T. An integrated Approach to Design Patterns Formalization, To appear in *Design Pattern Formalization Techniques*, Toufik Taibi (Ed), Idea Group Inc., Hershey, USA, 2007.

[12] M. Voelter, J. Noble and D. Manolesco (Editors), *Pattern Languages of Program Design*, Volume 5, Addison-Wesley, 2006.

[13] Vlissides, J.M., Multicast, *C++ Report*, Sep'1997.

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

[15] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[16] Zimmer, W., Relationships between design patterns. *In Pattern Languages of Program Design*. Addison-Wesley, 1994.