JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 8, No. 2, March–Aprile 2009

Stepwise Refinement Validation of Design Patterns Formalized in TLA+ using the TLC Model Checker

Toufik Taibi (ttaibi@uwo.ca) Department of Electrical and Computer Engineering University of Western Ontario London N6A5B9, Ontario, Canada
Ángel Herranz (aherranz@fi.upm.es), Babel Group, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, Boadilla del Monte 28660, Spain
Juan José Moreno-Navarro (jjmoreno@fi.upm.es), Babel Group, Universidad Politécnica de Madrid & IMDEA-Software, Campus de Montegancedo s/n, Boadilla del Monte 28660, Spain

Despite being around for only a little more than a decade, design patterns have proved to be successful reuse artifacts. However, the fact that they are mostly described informally gives rise to ambiguity and hinders correct usage. This paper discusses how to formally specify the "solution element" of patterns using TLA+, the formal specification language of Temporal Logic of Actions (TLA). The focus is first on formally specifying the most abstract version of a pattern before validly doing stepwise refinements by adding more details along the way until reaching a concrete implementation. Checking that the refinement properties hold was done using TLC — the TLA+ model checker.

Keywords: Design Patterns, Temporal Logic of Actions (TLA),Model Checking.

1 INTRODUCTION

A design pattern is a description of a set of successful solutions of a recurring problem within a context. A pattern is therefore made-of three pillars: a problem, a context and a solution [1]. Reusing patterns yields good quality designs as well as improves the productivity of designers.

Design patterns are mostly described using a combination of text, Unified Modeling Language (UML) [13] diagrams and sample code fragments. The intention is to make them easy to read and use, build a pattern vocabulary and a community of writers and users. However, these informal descriptions give rise to ambiguity, hinder correct usage, and limit tool support.

As such, formal specification of design patterns can complement informal ones by allowing rigorous reasoning about design patterns and facilitating tool support for their usage. Tool support could play a great role in automated pattern mining, detection of pattern variants, refactoring, code generation from specifications of patterns, or, simply checking if implementations adhere to properties of patterns.

Cite this article as follows: : *Stepwise Refinement Validation of Design Patterns Formalized in TLA+ using the TLC Model Checker*, in Journal of Object Technology, vol. 8, no. 2, March–Aprile 2009, pages 137–161,
http://www.jot.fm/issues/issues_2009_03/article3

In this paper, we present a formal framework to specify the "solution element" of a pattern. The solution element corresponds to the structure, participants and collaborations sections of the GoF catalog [5] description. These sections are the most coherent and as such the most valuable to formalize.

The framework uses Temporal Logic of Actions (TLA) as formal basis [10]. We will show that the framework is very expressive. In particular, patterns can be specified at different levels of abstraction. More importantly, these versions can be formally related through refinement, which is defined in the same framework. At the same time, validation through model checking will establish that a specification in a given level of abstraction is indeed a refinement of a specification of a higher level.

The rest of the paper is organized as follows. TLA is presented in Section 2. Section 3 presents our framework for formally specifying design patterns (using TLA) at different levels of abstraction as well as validating refinements relations between them.

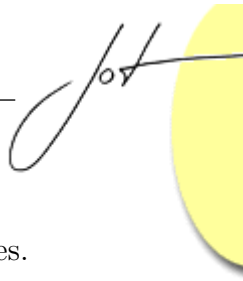
Section 4 introduces a case study that will act as a proof of concept for our proposed framework. In this case study, specifications have been written in TLA+ [11], which is a fully-fledged specification language based on TLA. The specifications are as follows:

- Our own abstract specification of the OBSERVER pattern (at least more abstract than the description of the pattern in [5]),
- A refinement of that specification that follows all details of the OBSERVER pattern as described in [5], and
- The specification of an instance (concrete implementation) of the OBSERVER pattern.

In Section 5 we use TLC (the TLA+ model checker), to check that the instance of the OBSERVER pattern is a refinement of the GoF version which is in turn a refinement of the most abstract version. Section 6 describes related work, while section 7 concludes the paper.

2 OVERVIEW OF TLA

In order to make the paper self-contained, this section provides a detailed description (with examples) of TLA. TLA [10] was developed for describing and reasoning about concurrent and distributed systems. It is essentially used to specify the *behavioral* properties of a system i.e what the system is supposed to do. TLA specifies a system by describing its allowed *behaviors* i.e what it may do in the course of an execution. A behavior is an infinite sequence of states.



A *step* is a pair of successive states in a behavior.

A *state* is a function from *variables* (or *flexible variables*) of the system to values.

Assuming a system with one variable x, $s_0 = \{x \mapsto 0\}$, $s_1 = \{x \mapsto 1\}$, $s_2 = \{x \mapsto$ "`abc`"$\}$ and $s_3 = \{x \mapsto red\}$ are states of the system.

A *state function* is a non-Boolean expression built from variables and constants of the system, i.e. a state function is a mapping from states to values. In our example, $x + 1$ is a state function that maps s_0 to 1, s_1 to 2 and s_2 and s_3 to an undefined value.

A *predicate* (or *state predicate*) is a Boolean expression built from variables and constants of the system, i.e. a predicate is a mapping from states to Booleans. In our example, $x \in Nat$ (*Nat* is the set of natural numbers) is a predicate that maps s_0 and s_1 to TRUE and s_2 and s_3 to FALSE.

An *action* is a Boolean expression built from variables, *primed variables* (flexible variables adorned with "′") and constants of the system. An action represents a relation between old states and new states, where unprimed variables refer to the old state and primed variables refer to the new state. Thus, an action is a mapping from pair of states to Booleans where the first state maps unprimed variables and the second state maps primed variables.

In our example, $x' = x + 1$ is an action that maps (s_0, s_1) to TRUE and any other pair of the previous example states to FALSE.

State functions and actions can be given a name in a *definition* (followed by the symbol $\triangleq$) and can be parametrized with *rigid variables* that denote fixed but unknown values, i.e. the values of rigid variables do not change and remain the same in the old and new state. *Quantification* ($\forall$ and $\exists$) over rigid variables is allowed. In our example, definition $Test(n) \triangleq n = x$ introduces a unary predicate named *Test* and action $A \triangleq \exists n \in Nat : x' = x + n$ relates states where x is greater or equal than its value in the previous state.

A *formula* (or *temporal formula*) is built from actions using logical connectives (basically $\wedge$ and $\neg$ as the others can be derived from these two), and the unary operator $\Box$ (*always*). Unary operator $\Diamond$ (*eventually*) can be derived from $\Box$ by $\neg\Box\neg F$.

Formulas are assertions about behaviors. Let $\sigma = \langle s_0, s_1, \ldots\rangle$ be a behaviour. σ satisfies an action A iff A maps (s_0, s_1) to TRUE. σ satisfies $F \wedge G$ iff σ satisfies F and σ satisfies G. σ satisfies $\neg F$ iff σ does not satisfies F. σ satisfies $\Box F$ if any behavior $\langle s_n, s_{n+1}, \ldots\rangle$ (n being a natural number) satisfies F.

In our example, formula $\Box(((x = 0) \wedge (x' = x + 1)) \vee ((x = 1) \wedge (x' = x - 1)))$ is satisfied by $\langle\{x \mapsto 0\}, \{x \mapsto 1\}, \{x \mapsto 0\}, \{x \mapsto 1\}, \ldots\rangle$ and by $\langle\{x \mapsto 1\}, \{x \mapsto 0\}, \{x \mapsto 1\}, \{x \mapsto 0\}, \ldots\rangle$ and it is not satisfied by $\langle\{x \mapsto 1\}, \{x \mapsto 1\}, \{x \mapsto 0\}, \{x \mapsto 0\}, \ldots\rangle$.

For any state function or predicate E, we define E' to be the expression ob-

tained by replacing each variable v in E by the primed variable v'. The expression UNCHANGED f means $f' = f$ (f is a state function).

Stuttering on action A under the vector of variables f occurs when either the action A occurs or the variables in f remain unchanged. The stuttering operator and its dual angle operator are defined as follows. $[A]_f \triangleq A \vee (f' = f)$ and $\langle A \rangle_f \triangleq A \wedge (f' \neq f)$. Stuttering allows to compose systems and interleave actions of different systems.

For any action A, ENABLED A is a predicate that is true for a state iff it is possible to take an A step starting in that state.

TLA recommends a style in which conjuncts and disjuncts are preceded by $\wedge$ or $\vee$ respectively. This eliminates the need for parenthesis making the notation especially useful when conjunctions and disjunctions are nested.

Specification are usually written to handle two types of properties of a system: *safety* and *liveness*. Safety properties ensure what a system must not do, while liveness properties ensure that something does happen. Safety is handled by the way specifications are written, which implicitly defines behaviors that could satisfy them. Liveness is handled through "explicit" fairness requirement.

TLA defines two types of fairness properties: *weak fairness* and *strong fairness* as follows:

- $\mathrm{WF}_f(A) \triangleq (\Box\Diamond\langle A \rangle_f) \vee ((\Box\Diamond\neg \mathit{Enabled}\langle A \rangle_f)$, which means that either infinitely many A steps occur or A is infinitely often disabled. In other words [11], an A step must eventually occur if A is repeatedly enabled without interruption.
- $\mathrm{SF}_f(A) \triangleq (\Box\Diamond\langle A \rangle_f) \vee ((\Diamond\Box\neg \mathit{Enabled}\langle A \rangle_f)$, which means that either infinitely many A steps occur or A is eventually disabled forever. In other words [11], an A step must eventually occur if A is repeatedly enable, possibly with interruptions.

In TLA, systems are represented as a tuple of variables f, a conjunction of an initial condition $\mathit{Init}_{\mathcal{S}}$, an action $\mathit{Next}_{\mathcal{S}}$ (which can be in fact a disjunction of actions $A_{i\mathcal{S}}$) that is continually repeated under stuttering, and a set of fairness conditions $\mathit{Fair}_{\mathcal{S}}$ (conjunction of 0 or more formulas of the form $\mathrm{WF}_f(A_{j\mathcal{S}})$ or $\mathrm{SF}_f(A_{j\mathcal{S}})$). Here $A_{j\mathcal{S}}$ represents a disjunction of actions in $\mathit{Next}_{\mathcal{S}}$. As such, TLA specifications can be written as $\mathcal{S} \triangleq \mathit{Init}_{\mathcal{S}} \wedge \Box[\mathit{Next}_{\mathcal{S}}]_f \wedge \mathit{Fair}_{\mathcal{S}}$. Theorems (proof obligations) of the form $\mathcal{S} \Rightarrow P$ can be added to the specification in order to establish that property P should be valid in specification $\mathcal{S}$.

To wrap-up this section, let us model a light switch and provide concrete examples of some concepts defined above.

$$
\begin{array}{ll}
\mathit{Invariant} \triangleq x \in \{0,1\} & \mathit{Next} \triangleq \mathit{On} \vee \mathit{Off} \\
\mathit{Init} \triangleq x = 0 & \mathit{Spec} \triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_S \wedge \mathrm{WF}_S(\mathit{Off}) \\
\mathit{On} \triangleq x = 0 \wedge x' = 1 & \text{THEOREM } \mathit{Spec} \Rightarrow \Box \mathit{Invariant} \\
\mathit{Off} \triangleq x = 1 \wedge x' = 0 &
\end{array}
$$

The above specification has only two possible states. The state where the light switch is off ($\{x \mapsto 0\}$ which is also the initial state) and the state where the light switch is on($\{x \mapsto 1\}$). $x' = 1$ is an example of an action that assigns the value 1 to x after action *On* is executed. The following are examples of behaviors that satisfy formula *Spec*:

$$\langle\{x \mapsto 0\}, \{x \mapsto 1\}, \{x \mapsto 0\}, \{x \mapsto 1\}, \{x \mapsto 1\}, \{x \mapsto 1\}, \{x \mapsto 0\}, \ldots\rangle$$
$$\langle\{x \mapsto 0\}, \{x \mapsto 0\}, \{x \mapsto 1\}, \{x \mapsto 1\}, \{x \mapsto 0\}, \{x \mapsto 0\}, \{x \mapsto 0\}, \ldots\rangle$$

The *weak fairness* condition stipulates that action *Off* should be executed infinitely often (provided it is enabled) as we wish the light to be off to save energy especially if there are many stuttering steps in which $x = 1$ and the light is not being used.

TLA possesses a fully-fledged language called TLA+ that allows the specification of the behavior of virtually any system. For years it has been successfully used to specify hardware systems and is gaining popularity in specifying software systems [11]. Moreover, TLA+ has a model checker named TLC that allows to check if a given model satisfies a given TLA formula as well as allowing the verification of the satisfiability of invariants and properties of the system.

3 A FRAMEWORK FOR THE FORMAL SPECIFICATION OF DESIGN PATTERNS

In this sections we introduce our framework which allows the specification of patterns at different levels of abstraction and the validation of the refinement relationships which exist between the different specifications.

Patterns Specification

The structural aspect of patterns is represented by sub-classes participating in the pattern and associations 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. Associations are represented as mathematical relations between objects.

In our framework, classes are defined as TLA+ constants, while associations between classes are defined as TLA+ flexible variables.

The behavioral aspect of a pattern is captured using actions which describe valid changes in the association between objects.

The structure of a TLA+ specification of a design pattern is shown in Figure 1. All TLA+ specifications shown in this paper have been well commented (in shaded gray) in order to make them as self-explanatory as possible. Moreover, TLA+

constructs used will not be detailed here. The reader is advised to see [11] for further details.

MODULE *Pattern*

CONSTANTS

$C1,\ C2,\ \ldots,\ Cq$ Classes as constant set of instances

VARIABLES

$u1,\ u2,\ \ldots,\ un$ Variables that represent associations of the pattern

---

$Invariants \triangleq I1 \land I2 \land \ldots \land Ik$ Invariants capture type and cardinality of associations

$Properties \triangleq P1 \land P2 \land \ldots \land Pl$ Properties of the pattern

---

$Init \triangleq P$ Initial predicate defines valid initial states

$Ai \triangleq \ldots$ Valid state transition, with i from 1 to m

$Next \triangleq A1 \lor A2 \lor \ldots \lor Am$ Valid state transitions defined as disjuntion of actions

$u \triangleq \langle u1,\ u2,\ \ldots,\ un \rangle$ A name for the tuple of variables

WF_$u(Bj)$/SF_$u(Bj)$ means either strong or weak fairness

Bj represent any disjunction of actions Ai

$F \triangleq \mathrm{WF}_u(B1)/\mathrm{SF}_u(B1) \land \ldots \land \mathrm{WF}_u(Bp)/\mathrm{SF}_u(Bp)$

$Spec \triangleq Init \land \Box[Next]_u \land F$ The specification of the pattern

---

Theorems are proof obligations that reflect that state changes described by actions preserve invariants and satisfy pattern properties.

THEOREM $Spec \Rightarrow \Box Invariants$ Pattern invariants must be always preserved

THEOREM $Spec \Rightarrow Properties$ Pattern properties must be satisfied

Figure 1: Structure of a TLA+ specification of patterns

The Refinement Process

The main advantage of our approach is that the focus is first given to specifying the most abstract version of a given pattern such that "low-level" programming details are avoided. In later versions of the specification, other "implementation-level" details can be "gradually" introduced. What should be kept and what should be left out in a higher level specification is pattern specific and is of course influenced by the experience of the specifier.

A design pattern Q is a refinement (or a lower-level version) of a design pattern P if every allowed behavior in Q is allowed in P [10]. If Q is specified using a TLA formula Ψ and P is specified using TLA formula Φ, Q is a refinement of P if Ψ is a refinement of Φ.

In order to formally define refinement, we need first to formally define the concept of "refinement mapping" [10]. If Δ is a TLA specification, let C_Δ be the set of

constants of Δ and V_Δ is the set of variables of Δ.

Definition 3.1 (Refinement mapping)**.** Let Ψ and Φ be two specifications and let $\rho : C_\Phi \cup V_\Phi \rightarrow C_\Psi \cup V_\Psi$. ρ is a **refinement mapping** from Ψ to Φ iff ρ is a total function and $\Psi \Rightarrow \rho(\Phi)$. $\rho(\Phi)$ represents the substitution of constants and variables of Φ by those of Ψ.

Definition 3.2 (Refinement)**.** Let Ψ and Φ be two specifications. Ψ is a **refinement** of Φ if there exists a refinement mapping from Φ to Ψ.

As shown in Figure 2, we must explicitly relate states in the concrete specification with states in abstract specifications and this can be done in the form of substitutions (or *refinement mappings*) of constant and flexible variables of the abstract specification with those of the concrete specification.

This technique is used in Section 5 to validate that the GoF version of the OBSERVER pattern in [5] is a refinement of our abstract version of the same pattern and to check that a concrete implementation is an instance of the GoF version of the OBSERVER pattern.

MODULE *QrefinesP*

Concrete design pattern
EXTENDS Q

Assume P has m constants ($C1, \ldots, Cm$) and n variables ($x1, \ldots, xn$)
Assume Q has p constants ($K1, \ldots, Kp$) and q variables ($y1, \ldots, yq$)
$f1, \ldots, fm$ and $g1, \ldots, gn$ are refinement mappings
$Abstract \triangleq$ INSTANCE P WITH
$C1 \leftarrow f1(K1, \ldots, Kp)$,
$\ldots$,
$Cm \leftarrow fm(K1, \ldots, Kp)$,
$x1 \leftarrow g1(K1, \ldots, Kp, y1, \ldots, yq)$,
$\ldots$,
$xn \leftarrow gn(K1, \ldots, Kp, y1, \ldots, yq)$

$AbstractSpec \triangleq Abstract!Spec$ This is P's specification

THEOREM $Spec \Rightarrow AbstractSpec$ Spec is Q's specification

Figure 2: Structure of a TLA+ refinement of patterns

4 CASE STUDY

Our case study includes the specification of an abstract version of the OBSERVER pattern, the specification of the GoF version of the same pattern and the specification

of an instance(concrete implementation) of the same pattern.

Abstract Version of the OBSERVER Pattern

In the OBSERVER pattern [5] there are concrete observers and concrete subjects. A concrete subject has data (attributes) whose values can be modified. Concrete observers can be interested in changes that occur in a concrete subject's data. The pattern describes how concrete subjects and concrete observers are connected with each other and how they communicate in order to preserve data consistency. A concrete subject notifies its concrete observers whenever a change occurs that could make their data inconsistent with its own. After being notified of the change, concrete observers query a concrete subject to get the latest values of its data.

Figure 3 depicts the UML class diagram of an abstract version of the OBSERVER pattern. This version is more abstract than the one appearing in [5] because the focus is only on modeling and specifying the relations between participating classes rather than on their attributes. As such, details of attributes and methods appearing in [5] have been omitted from Figure 3 because they are not needed at this level of abstraction.

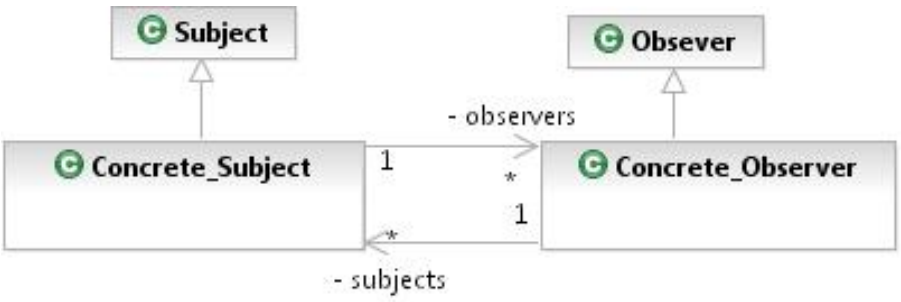


Figure 3: UML class diagram of the most abstract version of the OBSERVER pattern

Figures 4, 5 and 6 depict the TLA+ specifications of the abstract version of the OBSERVER pattern. The specification was split into three modules for reasons of space and clarity: *ObserverStruct*, *ObserverBehav* and *Observer*.

In module *ObserverStruct* (Figure 4), classes Concrete_Subject and Concrete_Observer are defined as TLA+ constants *ConcreteSubject* and *ConcreteObserver* respectively. Two associations between concrete subjects and concrete observers have been introduced. The first one handles the case of a concrete observer attached to a concrete subject, while the second handles the case of a concrete observer updated after a state change in its attached concrete subject. These associations are represented by TLA+ variables (mathematical relations) *attached* and *updated* respectively. In module *ObserverBehav* (Figure 5), predicate $(s, o) \in attached$ indicates that o is attached to s and $(s, o) \in updated$ indicates that the data of o is consistent with the data of s.

MODULE *ObserverStruct*

CONSTANT *ConcreteSubject*, *ConcreteObserver*
VARIABLE *attached*, *updated*

Auxiliary definitions

$u \triangleq \langle attached, updated \rangle$
$CS\_CO \triangleq ConcreteSubject \times ConcreteObserver$
$attached\_to(s) \triangleq \{y \in ConcreteObserver : \langle s, y \rangle \in attached\}$
$updated\_wrt(s) \triangleq \{y \in ConcreteObserver : \langle s, y \rangle \in updated\}$

$Invariant \triangleq attached \subseteq CS\_CO \wedge updated \subseteq attached$

$Init \triangleq attached = \{\} \wedge updated = \{\}$

Figure 4: TLA+ module *ObserverSruct*

Module *ObserverStruct* captures the above properties through an invariant that establishes that *attached* and *updated* are subsets of the Cartesian product *ConcreteSubject*× *ConcreteObserver* (type definition) and that *updated* is always a subset of *attached*. The initial state predicate *Init* initializes the relations *attached* and *updated* as empty relations.

Behavioral requirements of the abstract version of the OBSERVER pattern are captured in module *ObserverBehav* (Figure 5) as follows:

- A concrete observer o can attach to a concrete subject s showing that o is interested in changes in the data of s. This is reflected by action $Attach(s, o)$.
- A change in data occurs in a concrete subject s. This is reflected by action $Set\_state(s)$.
- Based on the above, all concrete observers attached to a concrete subject s should have their data updated in order to be consistent with the one of s. This is reflected by action *Update*.
- A concrete observer o can detach from a concrete subject s showing that it is no longer interested in its data change. This is reflected by action $Detach(s, o)$.

Finally, module *Observer* (Figure 6) provides the complete specification of this abstract version of the OBSERVER pattern. Since the action to be executed is selected non-deterministically (provided that its precondition is true), a *weak fairness* requirement was introduced. It states that the action *Update* should be executed infinitely often. The theorem reflects that the execution of the actions must preserve the invariant.

MODULE *ObserverBehav*

EXTENDS *ObserverStruct*

A concrete observer o can attach to a concrete subject s (not previously attached to) showing that o is interested in changes in the data of s.

$$
\begin{array}{l}
Attach(s,\ o) \triangleq \\
\quad \land \langle s,\ o\rangle \notin attached \\
\quad \land attached' = attached \cup \{\langle s,\ o\rangle\} \land updated' = updated \cup \{\langle s,\ o\rangle\}
\end{array}
$$

A change in subject s occurs once observer were notified about previous change and then attached observers became not updated.

$$
\begin{array}{l}
Set\_state(s) \triangleq \\
\quad \land attached\_to(s) = updated\_wrt(s) \land attached\_to(s) \neq \{\} \\
\quad \land updated' = updated \setminus \{\langle x,\ y\rangle \in CS\_CO : x = s\} \\
\quad \land \text{UNCHANGED } attached
\end{array}
$$

Observers attached to a subject s and not yet notified about a change should have their data updated in order to be consistent with the one of s.

$$
\begin{array}{l}
Update \triangleq \\
\quad \land \exists\, s \in ConcreteSubject : \\
\qquad (\land attached\_to(s) \neq updated\_wrt(s) \\
\qquad\ \land updated' = updated \cup \{\langle x,\ y\rangle \in attached : x = s\}) \\
\quad \land \text{UNCHANGED } attached
\end{array}
$$

A concrete (already updated) observer o can detach from a concrete subject s showing that it is no longer interested in its data change.

$$
\begin{array}{l}
Detach(s,\ o) \triangleq \\
\quad \land \langle s,\ o\rangle \in attached \\
\quad \land \langle s,\ o\rangle \in updated \\
\quad \land attached' = attached \setminus \{\langle s,\ o\rangle\} \\
\quad \land updated' = updated \setminus \{\langle s,\ o\rangle\}
\end{array}
$$

Figure 5: TLA+ module *ObserverBehav*

MODULE *Observer*

EXTENDS *ObserverBehav*

$$
\begin{array}{l}
Next \triangleq \exists\, s \in ConcreteSubject,\ o \in ConcreteObserver : \\
\qquad \lor Attach(s,\ o) \lor Set\_state(s) \lor Update \lor Detach(s,\ o) \\
Liveness \triangleq \mathrm{WF}_u(Update) \\
Spec \triangleq Init \land \Box[Next]_u \land Liveness
\end{array}
$$

THEOREM $Spec \Rightarrow \Box Invariant$

Figure 6: TLA+ module *Observer*

Model Checking Specifications Using TLC

TLA+ models can be validated in order to make sure that a formula is satisfied by all behaviors of a given system. Model checkers can explore behaviors allowed by the model, possibly detecting deadlock or violation of invariants.

TLC [11] is a model checker for specifications written in TLA+. TLC is an explicit-state, on-the-fly model checker. TLA+ and TLC have been successfully used in many practical situations, particularly by hardware engineers to check the correctness of hardware protocols. It is being gradually used by software engineers to specify and check concurrent algorithms and protocols for software systems.

TLC requires a configuration file that defines the finite-state model and the formula that represent the system specification to analyze and the properties to check. Let us have a look at the self-explanatory configuration file for our module *Observer*:

With this file, TLC is instructed to check $Spec \Rightarrow \Box Invariant$ (indicated by clauses `SPECIFICATION` and `INVARIANTS` in a model where *ConcreteSubject* and *ConcreteObserver* are defined as sets $\{s1, s2\}$ and $\{o1, o2, o3, o4\}$ respectively, where elements represent different objects).

TLC firsts checks the syntactic and semantic correctness and well-formedness 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. TLC also reports the number of states it generated during its analysis, the number of distinct states, and the depth of the state graph (the length of the longest path). For small models TLC run completes after few seconds. Trying to analyze somewhat larger models, leads to the well-known problem of state-space explosion.

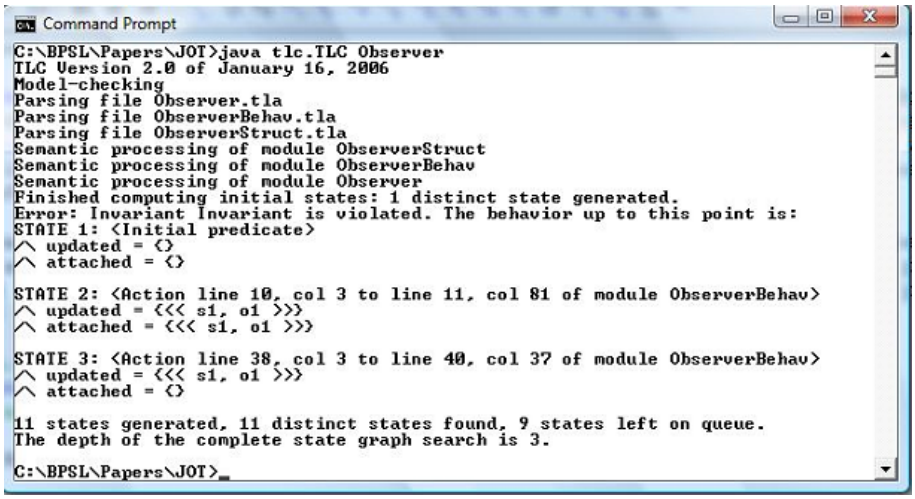


Figure 7: TLC detecting an invariant violation in the *Observer* module

To show how TLC can detect errors, let's re-define action $Detach(s, o)$ in module

ObserverBehav as follows:

$Detach(s, o) \triangleq$
$\land \langle s, o \rangle \in attached$
$\land attached' = attached \setminus \{\langle s, o \rangle\}$
$\land UNCHANGED\ updated$

The TLC output in Figure 7 indicates an invariant violation. More precisely, $updated \subseteq attached$ is not preserved by action *Detach*. The problem was that the precondition of *Detach* is too weak because $\langle s, o \rangle$ should belong to relation *updated* and because such relation must be changed by removing the detached pair.

GoF Version of the Observer Pattern

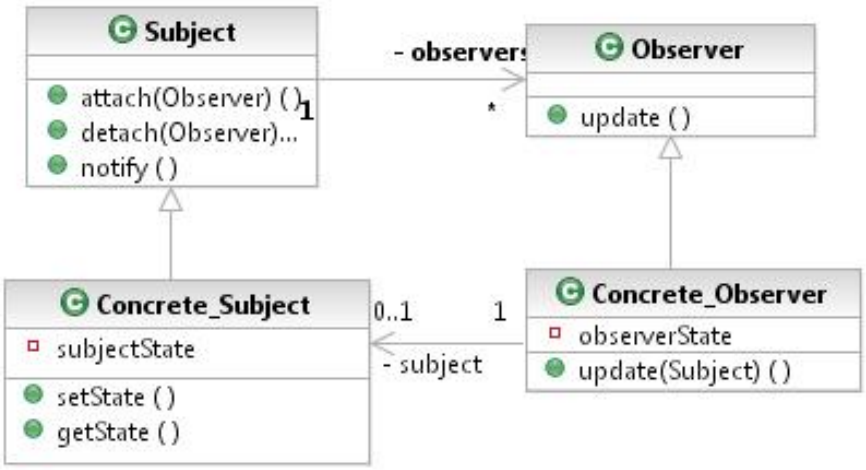


Figure 8: UML class diagram of the GoF version of the Observer pattern

Figure 8 shows the UML class diagram of the GoF version of the Observer pattern [5]. Again, the TLA+ specification of this version of the pattern has been divided into three parts. The structural part represents classes, attributes, invariants and the initial state (Figure 9 depicting module *ObserverGOFStruct*). The behavioral part shows allowed actions (Figure 10 depicting module *ObserverGOFBehav*). The last part defines the TLA+ formula specifying the pattern (Figure 11 depicting module *ObserverGOF*).

Below is a list of changes introduced in the specification *ObserverGOFStruct* (Figure 9) as compared to *ObserverStruct* defined in the previous subsection:

- Instead of using mathematical relations *attached* and *updated*, a concrete subject maintains an attribute called *observers* representing a sequence of concrete observers attached to it, while each concrete observer maintains an attribute

MODULE *ObserverGOFStruct*

LOCAL INSTANCE *Naturals*
LOCAL INSTANCE *Sequences*
CONSTANT *Observer*, *Subject*, *Data* Classes
VARIABLE
observers, *subject*, Attributes that keep structural relations
subjectState, *observerState* Internal state representation

Auxiliary definitions

$u \triangleq \langle subject,\ observers,\ subjectState,\ observerState \rangle$

Element x in sequence s

$in\_seq(x,\ s) \triangleq \exists\, i \in \text{DOMAIN } s : s[i] = x$

Set of observers attached to a subject s according to subject attribute

$observer\_list\_according\_to\_subject\_attribute(s) \triangleq$
$\{x \in \text{DOMAIN } subject : in\_seq(s,\ subject[x])\}$

Set of observers attached to a subject s according to observers attribute

$observers\_list\_according\_to\_observers\_attribute(s) \triangleq$
$\{x \in Observer : in\_seq(x,\ observers[s])\}$

Predicate that establishes subject s and observer o are attached

$attached\_P(s,\ o) \triangleq in\_seq(o,\ observers[s]) \land in\_seq(s,\ subject[o])$

Predicate that establishes observer o is updated wrt subject s

$updated\_P(s,\ o) \triangleq attached\_P(s,\ o) \land subjectState[s] = observerState[o]$

Set of observers attached to subject s

$attached\_to(s) \triangleq \{x \in Observer : attached\_P(s,\ x)\}$

Set of observers updated wrt to subject s

$updated\_wrt(s) \triangleq \{x \in attached\_to(s) : updated\_P(s,\ x)\}$

Invariant and properties

$Types \triangleq$
$\land\ subject \in [Observer \to Seq(Subject)]$
$\land\ observers \in [Subject \to Seq(Observer)]$
$\land\ subjectState \in [Subject \to Data]$
$\land\ observerState \in [Observer \to Data]$
$Cardinality \triangleq \forall\, x \in \text{DOMAIN } subject : Len(subject[x]) \leq 1$
$Invariant \triangleq$
$\land\ Types \land Cardinality$

No repeated observers in the observers field of every subject

$\land \forall\, s \in \text{DOMAIN } observers :$
$\quad \forall\, i, j \in \text{DOMAIN } observers[s] :$
$\quad\quad observers[s][i] = observers[s][j] \Rightarrow i = j$

subject and observers fields must be consistent

$\land \forall\, x \in Subject :$
$\quad observer\_list\_according\_to\_subject\_attribute(x)$
$\quad = observers\_list\_according\_to\_observers\_attribute(x)$

$Init \triangleq$
$\land\ subject = [x \in Observer \mapsto \langle\rangle]$
$\land\ observers = [x \in Subject \mapsto \langle\rangle]$
$\land\ subjectState = [x \in Subject \mapsto 0]$
$\land\ observerState = [x \in Observer \mapsto 0]$

Figure 9: TLA+ module *ObserverGOFStruct*

called *subject* with a sequence containing no more than a unique concrete subject to which it is attached.

- State change in concrete subjects and concrete observers is explicitly shown using the attributes *subjectState* and *observerState*. The set *Data* has been introduced to contain any valid values for *subjectState* and *observerState*.
- All above four attributes are represented as TLA+ variables that have been defined as functions.
- The specification now defines an invariant containing four conjuncts. The first is about the type of TLA variables used. The second is in fact a constraint that concrete observers should be attached to at most one concrete subject. The third ensures that the list of concrete observers maintained by a concrete subject does not contain duplication. Finally the last ensures consistency between the list of concrete observers attached to a concrete subject and the concrete subject to which a concrete observer is attached to.

With respect to the behavioral part of the specification (Figure 10), actions have been adapted to the newly introduced variables but semantically they achieve the same purpose as in the previous version. This refinement step will be validated in Section 5 by using TLC.

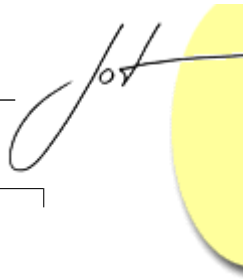
An Instance of the Observer Pattern

Figure 12 shows the UML class diagram of an instance (concrete implementation) of the Observer pattern. It appeared in [5] in the sample code section of the Observer pattern. ClockTimer is concrete subject for storing and maintaining the time of the day. It notifies its concrete observers after every tick. ClockTimer provides a method for retrieving the time. The tick() method gets called by an internal timer at regular intervals. Method tick () updates the Clock Timer's internal state and calls method notify() to inform concrete observers of the change.

The classes DigitalClock and AnalogClock are concrete observers used to display the time in a digital and analog fashion respectively. When time ticks, the two clocks will be updated and will re-display themselves appropriately. The TLA+ specification of this instance of the Observer pattern can be found in Appendix A (module *ClockObserver*) and it is very similar to the specifications in Figures 9, 10 and 11 with the following substitutions: *Subject* becomes *ClockTimer*, *Observer* becomes *Clock* (which is $DigitalClock \cup AnalogClock$), *subjectState* becomes *subjectTime* and *observerState* becomes *observerTime*.

5 STEPWISE REFINEMENT VALIDATION USING TLC

In this section we show how to validate the refinement between the three versions of the Observer pattern. Figure 13 and Figure 14 depict refinement mappings written to validate that the GoF version of the Observer pattern is a refinement



```
MODULE ObserverGOFBehav
EXTENDS ObserverGOFStruct
LOCAL INSTANCE Naturals
LOCAL INSTANCE Sequences

Attach(s, o) ≜
  ∧ ¬attached_P(s, o) ∧ Len(subject[o]) = 0
  ∧ observers' = [observers EXCEPT ![s] = Append(@, o)]
  ∧ subject' = [subject EXCEPT ![o] = ⟨s⟩]
  ∧ UNCHANGED subjectState
  ∧ observerState' = [observerState EXCEPT ![o] = subjectState[s]]

Set_state(s) ≜
  ∧ attached_to(s) = updated_wrt(s) ∧ attached_to(s) ≠ {}
  ∧ subjectState' = [subjectState EXCEPT ![s] = CHOOSE d ∈ Data : d ≠ @]
  ∧ UNCHANGED subject ∧ UNCHANGED observers ∧ UNCHANGED observerState

Update ≜
  ∧ ∃ s ∈ Subject :
      ( ∧ attached_to(s) ≠ updated_wrt(s)
        ∧ observerState' = [x ∈ Observer ↦
                              IF attached_P(s, x)
                               THEN subjectState[s]
                               ELSE observerState[x]])
  ∧ UNCHANGED subject ∧ UNCHANGED observers ∧ UNCHANGED subjectState

Detach(s, o) ≜
  ∧ attached_P(s, o) ∧ updated_P(s, o)
  ∧ observers' = [observers EXCEPT ![s] =
                    LET DifFromO(x) ≜ x ≠ o
                    IN  SelectSeq(@, DifFromO)]
  ∧ subject' = [subject EXCEPT ![o] = ⟨⟩]
  ∧ UNCHANGED subjectState ∧ UNCHANGED observerState
```

Figure 10: TLA+ module *ObserverGOFBehav*

of the abstract version and that the instance of the GoF version (*ClockObserver*) is in turn a refinement of the GoF version.

Figure 15 and Figure 16 show the outputs of running TLC on the TLA+ specifications given in Figure 13 and Figure 14 respectively. Both were without errors and showing indeed that the concrete implementation is a refinement of the GoF version of the OBSERVER pattern and that the later is in turn a refinement of the more abstract version.

MODULE *ObserverGOF*

EXTENDS *ObserverGOFBehav*

$$Next \triangleq \exists\, s \in Subject,\ o \in Observer : \\ \lor\ Attach(s,\ o) \lor Set\_state(s) \lor Update \lor Detach(s,\ o)$$
$$Liveness \triangleq \mathrm{WF}_u(Update)$$
$$Spec \triangleq\ Init \land \Box[Next]_u \land Liveness$$

THEOREM $Spec \Rightarrow \Box Invariant$

Figure 11: TLA+ module *ObserverGOF*

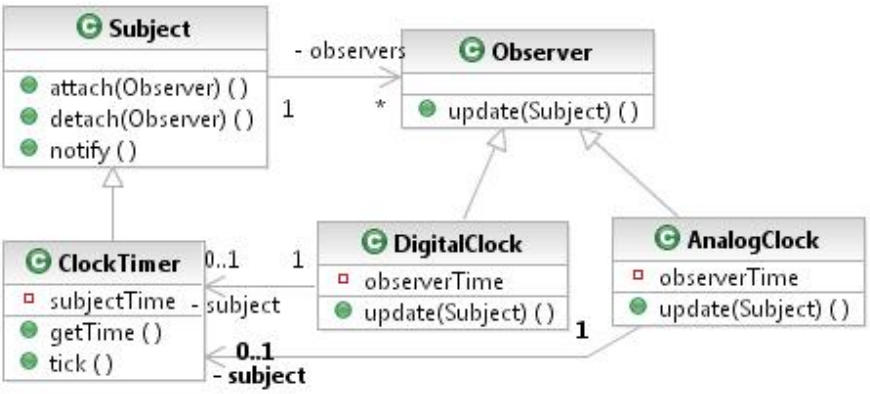


Figure 12: UML class diagram of an instance of the OBSERVER pattern

6 RELATED WORK

The approach presented in this paper builds on our previous work on Balanced Pattern Specification Language (BPSL) [16], a language which used First-Order Logic (FOL) [8] and Temporal Logic of Actions (TLA) [10] as formal basis to formally specify the structural and behavioral aspects of patterns respectively. In this new version we solely use TLA+ to specify both aspects of patterns. Additionally, we have developed a framework in which patterns can be specified at different levels of abstraction and techniques (using model checking) to validate the existence of refinement relationships. Furthermore, we can now automatically validate that a given implementation is indeed an instance of a given pattern.

[15] provided an in-depth description of 16 different design pattern formalization techniques. In this section we will summarize the features of the techniques that are the closest to our approach.

[12] describes Design Pattern Modeling Language (DPML), a notation supporting the specification of design pattern solutions and their instantiation into UML design models. DPML uses a simple set of visual abstractions and readily lends itself

MODULE *ObserverGOFRefinesObserver*

ObserverGOFRefinesObserver.tla - Specification of the validation that *ObserverGOF* specification refines the absract *Observer* pattern specification.

Refinement specification

EXTENDS *ObserverGOF*

Abstract specification. ← represents substitution of variables

Abstract $\triangleq$ INSTANCE *Observer* WITH
ConcreteSubject ← *Subject*,
ConcreteObserver ← *Observer*,
attached ← $\{\langle x, y\rangle \in Subject \times Observer : attached\_P(x, y)\}$,
updated ← $\{\langle x, y\rangle \in Subject \times Observer : updated\_P(x, y)\}$

AbstractSpec $\triangleq$ *Abstract*!*Spec*

The theorem reflects the refinement property to be checked between the two versions of patterns

THEOREM *Spec* ⇒ *AbstractSpec*

Figure 13: TLA+ Module *ObserverGOFRefinesObserver*

MODULE *ClockObserverRefinesObserverGOF*

Refinement *Specifiction*

EXTENDS *ClockObserver*

Abstract Specification. ← represents substitution of variables

OGOF $\triangleq$ INSTANCE *ObserverGOF* WITH
Subject ← *ClockTimer*,
Observer ← *Clock*,
subjectState ← *subjectTime*,
observerState ← *observerTime*

OGOF_Spec $\triangleq$ *OGOF*!*Spec*

The theorem reflects the refinement property to be checked between the two versions of patterns

THEOREM *Spec* ⇒ *OGOF_Spec*

Figure 14: TLA+ Module *ClockObserverRefinesObserverGOF*

to tool support. DPML design pattern solution specifications are used to construct visual, formal specifications of design patterns. DPML instantiation diagrams are used to link a design pattern solution specification to instances of a UML model, indicating the roles played by different UML elements in the generic design pattern solution. The main drawback of this approach is being based on UML meta-modeling techniques which can be described as semi-formal at best.

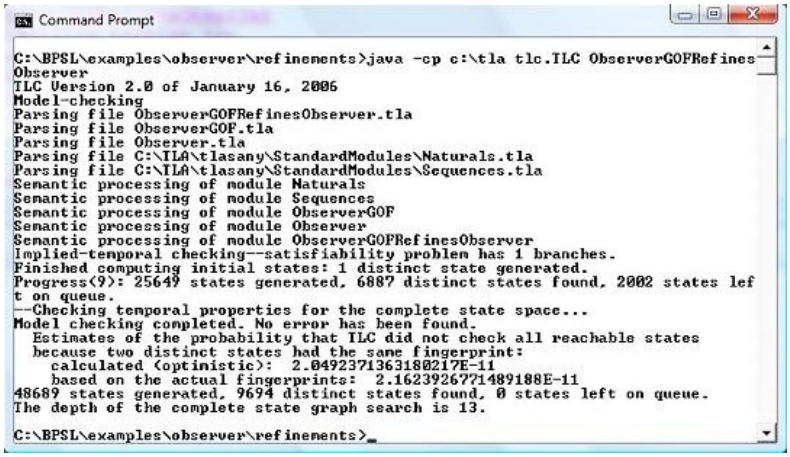


Figure 15: Running TLC on the *ObserverGOFRefinesObserver* with the shown configuration file

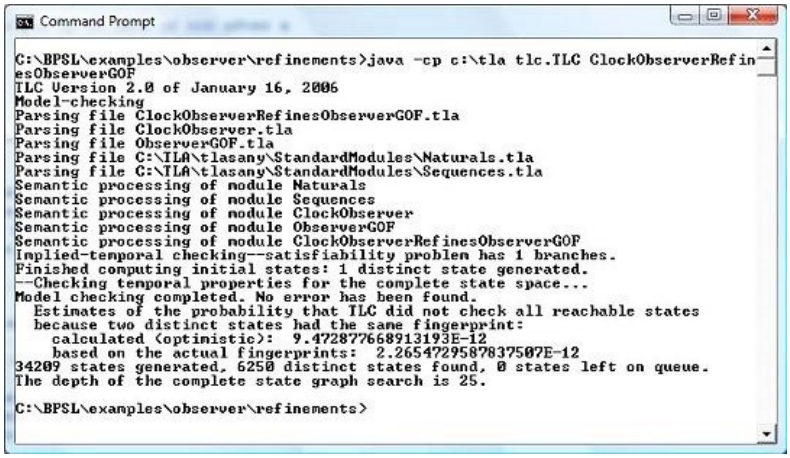
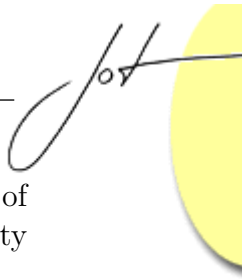


Figure 16: Running TLC on the *ClockObserverRefinesObserverGOF* module with the shown configuration file

[4] shows how formal specifications of GoF patterns, based on the Rigorous Approach to Industrial Software Engineering (RAISE) language, have been helpful to develop tool support. Thus, the OO design process is extended by the inclusion of pattern-based modeling and verification steps. The latter involving checking design correctness and appropriate pattern application through the use of a supporting tool, called DePMoVe (Design and Pattern Modeling and Verification). The main drawback of this approach is again its heavy reliance on meta-modeling techniques based on the RAISE language.

[6] describes an abstraction mechanism for collective behavior in reactive distributed systems. The mechanism allows to express recurring patterns of object interactions in a parametric form, and to formally verify temporal safety properties induced by applications of the patterns. The authors, discuss how the emphasis on full formality affects what can be expressed and achieved in terms of patterns of ob-



ject interactions. This approach focuses more on specifying the behavioral aspect of patterns than on their structural aspect. We believe that both aspects are equality important and deserve an equal attention.

[2] separated the structural and behavioral aspects of design patterns and proposed specification methods based on first-order logic, temporal logic, temporal logic of action, process calculus, and Prolog. It also explored verification techniques based on theorem proving. This approach lacks the integration between the specification of structural and behavioral aspects of patterns.

[9] describes a UML-based pattern specification language called Role-Based Meta-modeling Language (RBML) which defines the solution domain of design patterns in terms of roles at the meta-model level. This work discusses benefits of the RBML and presents notation for capturing various perspectives of pattern properties. As mentioned before, this approach suffers from the same weaknesses of any technique based on UML meta-models.

In [7], the formal specification of a design pattern is given as a class operator that transform a design given as a set of classes into a new design that takes into account the description and properties of the design pattern. The operator is specified in the SLAM-SL specification language, in terms of pre and post-conditions. Precondition collects properties required to apply the pattern and post-condition relates input classes and result classes encompassing most of the intent and consequences sections of the pattern. Many ideas of this work have been applied to our framework. However, although SLAM-SL is an executable language, it does not provide support for validating refinements.

[3] presents LePUS, a formal language for modeling OO design patterns. The authors demonstrated the language's unique efficacy in producing precise, concise, scalable, generic and appropriately abstract specifications modeling the GoF design patterns. Mathematical logic is used as a main frame of reference: the language is defined as a subset of first-order predicate calculus and implementations (programs) are modeled as finite structures in model theory. The main drawback of LePUS is that it focuses more on specifying the structural aspect of patterns than on their behavioral aspect.

Our approach is based on specifying both aspects (structural and behavioral) of patterns using one formalism i.e TLA. Moreover, we start by specifying the most abstract version of patterns and follow a stepwise refinement approach to formally specify other versions of the pattern up-to concrete implementations. We believe that specifying patterns at the highest level of abstraction first, make them easy to understand and hence easy to use. Furthermore, with our approach we were able to formally specify pattern composition [14]. We have used TLA+ to specify many GoF patterns (including their stepwise refinements) and we are in the process of completing the catalog.

7 CONCLUSION

Design patterns are means of improving design quality, flexibility and productivity. However, their inherent benefits cannot be fully exploited by the existing informal means of specification. Formal specification of patterns brings accuracy and facilitates tool support for their application.

In this paper, we defined a framework that uses TLA+ to formally specify patterns at different levels of abstractions. This has facilitated their understandability and hence their usability. The framework uses stepwise refinement to incrementally and validly add details to a specification after starting from the most abstract one. TLC was used to check the correctness of TLA+ specifications as well as the satisfiability of invariants and properties (defining the validation of refinement). The applicability of our approach was shown using three versions of the OBSERVER patterns as a case study.

In summary, the framework encompasses three characteristics:

- It allows the specification of the most abstract version of a pattern.
- It allows the specification of refinements or more concrete versions
- It allows the automatic validation of coherence of pattern descriptions as well as the refinement of patterns.

We are now in the process of defining (in TLA+) object creation and destruction in order to specify creational patterns. Also, we are in the process of defining a domain specific (subset of TLA+) language to describe patterns. This will allow us to develop tools to check the syntax and semantics of pattern specification and allows the forward and reverse engineering of design patterns and their specifications.

REFERENCES

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, 1978.

[2] J. Dong, P. Allencar, and D. Cowan. Formal specification and verification of design patterns. In T. Taibi, editor, *Design Patterns Formalization Techniques*, pages 94–108. Idea Group Inc., Hershey, USA, 2007.

[3] A.H Eden, E. Gasparis, and J. Nicholson. The gang of four companion: Formal specification of design patterns in lepus3 and class-z. Technical Report CSM-472, Department of Computer Science, University of Essex, 2007.

[4] A. Flores, A. Cechich, and G. Aranda. A generic model of object-oriented patterns specified in rsl. In T. Taibi, editor, *Design Patterns Formalization Techniques*, pages 44–72. Idea Group Inc., Hershey, USA, 2007.

[5] E. Gamma, R. Helm, R. Johnson, and G. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] J. Helin, P. Kellomaki, and T. Mikkonen. Patterns of collective behavior in ocsid. In T. Taibi, editor, *Design Patterns Formalization Techniques*, pages 73–92. Idea Group Inc., Hershey, USA, 2007.

[7] A. Herranz and J.J. Moreno-Navarro. Modeling and reasoning about design patterns in slam-sl. In T. Taibi, editor, *Design Patterns Formalization Techniques*, pages 206–235. Idea Group Inc., Hershey, USA, 2007.

[8] M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004.

[9] D.K Kim. The role-based metamodeling language for specifying design patterns. In T. Taibi, editor, *Design Patterns Formalization Techniques*, pages 183–205. Idea Group Inc., Hershey, USA, 2007.

[10] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[11] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[12] D. Mapelsden, J.G. Hosking, and J.C. Grundy. A visual language for design pattern modelling and instantiation. In T. Taibi, editor, *Design Patterns Formalization Techniques*, pages 20–43. Idea Group Inc., Hershey, USA, 2007.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[14] T. Taibi. Formalizing design patterns composition. *IEE Proceedings Software*, 153(3):127–136, 2006.

[15] T. Taibi, editor. *Design Patterns Formalization Techniques*. Idea Group Inc., Hershey, USA, 2007.

[16] T. Taibi and D.C.L Ngo. Formal specification of design patterns-a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

A TLA SPECIFICATION OF THE INSTANCE OF OBSERVER

This is the TLA+ specification of the instance of the OBSERVER pattern shown in Figure 12.

MODULE *ClockObserver*

ClockObserver.tla– Specification of an instance of the Observer pattern. This file is very similar to *ObseverGOF.tla* expect that the concrete subject is *ClockTimer* and the concrete observers are *DigitalClock* and *AnalogClock*.

LOCAL INSTANCE *Naturals*

LOCAL INSTANCE *Sequences*

CONSTANT

Classes of the system

ClockTimer, *DigitalClock*, *AnalogClock*, *Data*

VARIABLE

Structural relations: associations in the class diagram

subject, *DigitalClock*'s and *AnalogClock*'s field

observers, *ClockTimer*'s field

How to represent that observers are up-to-date or not after a change of their subject state?

subjectTime,

observerTime

$Clock \triangleq DigitalClock \cup AnalogClock$

$Types \triangleq$

$\land\ subject \in [Clock \to Seq(ClockTimer)]$

$\land\ observers \in [ClockTimer \to Seq(Clock)]$

$\land\ subjectTime \in [ClockTimer \to Data]$

$\land\ observerTime \in [Clock \to Data]$

$Cardinality \triangleq$

$\land \forall\, o \in \text{DOMAIN } subject : Len(subject[o]) \leq 1$

$InSeq(x,\ s) \triangleq \exists\, i \in \text{DOMAIN } s : s[i] = x$

Definitions used in the specification

$ObserverListAccordingToSubjectAttribute(s) \triangleq$

$\{x \in \text{DOMAIN } subject : InSeq(s, subject[x])\}$

$ObserversListAccordingToObserversAttribute(s) \triangleq$

$\{x \in DigitalClock \cup AnalogClock : InSeq(x, observers[s])\}$

$AttachedP(s, o) \triangleq$

$\land InSeq(o, observers[s])$

$\land InSeq(s, subject[o])$

$UpdatedP(s, o) \triangleq$

$\land subjectTime[s] = observerTime[o]$

$Attached\_List(s) \triangleq$

$\{x \in Clock : AttachedP(s, x)\}$

$Updated\_List(s) \triangleq$

$\{x \in Clock : UpdatedP(s, x)\}$

$$\begin{aligned} Invariant \triangleq \quad & \land Types \\ & \land Cardinality \\ & \land \forall x \in ClockTimer : ObserverListAccordingToSubjectAttribute(x) \\ & \qquad = ObserversListAccordingToObserversAttribute(x) \end{aligned}$$

$Init \triangleq$

$\land subject = [x \in Clock \mapsto \langle\rangle]$

$\land observers = [x \in ClockTimer \mapsto \langle\rangle]$

$\land subjectTime = [x \in ClockTimer \mapsto 0]$

$\land observerTime = [x \in Clock \mapsto 0]$

$Attach(s, o) \triangleq$

$\land \neg AttachedP(s, o)$

$\land Len(subject[o]) = 0$

$\land observers' = [observers \text{ EXCEPT } ![s] = Append(@, o)]$

$\land subject' = [subject \text{ EXCEPT } ![o] = \langle s \rangle]$

$\land \text{UNCHANGED } subjectTime$

$\land observerTime' = [observerTime \text{ EXCEPT } ![o] = subjectTime[s]]$

$Set\_state(s) \triangleq$

$\land Attached\_List(s) = Updated\_List(s)$

$\land Attached\_List(s) \neq \{\}$

$\land subjectTime' = [subjectTime \text{ EXCEPT } ![s] = \text{CHOOSE } d \in Data : d \neq @]$

$\land$ UNCHANGED $subject$

$\land$ UNCHANGED $observers$

$\land$ UNCHANGED $observerTime$

$Update \triangleq$

$\land \exists\, s \in ClockTimer :$

$(\land Attached\_List(s) \neq Updated\_List(s)$

$\land observerTime' = [x \in Clock \mapsto$

IF $AttachedP(s,\, x)$

THEN $subjectTime[s]$

ELSE $observerTime[x]])$

$\land$ UNCHANGED $subject$

$\land$ UNCHANGED $observers$

$\land$ UNCHANGED $subjectTime$

$Detach(s,\, o) \triangleq$

$\land AttachedP(s,\, o)$

$\land UpdatedP(s,\, o)$

$\land observers' = [observers$ EXCEPT $![s] =$

LET $DifFromO(x) \triangleq x \neq o$

IN $SelectSeq(@,\, DifFromO)]$

$\land subject' = [subject$ EXCEPT $![o] = \langle\rangle]$

$\land$ UNCHANGED $subjectTime$

$\land$ UNCHANGED $observerTime$

$Next \triangleq$

$\lor (\exists\, s \in ClockTimer,\, o \in Clock : Attach(s,\, o))$

$\lor (\exists\, s \in ClockTimer : Set\_state(s))$

$\lor Update$

$\lor (\exists\, s \in ClockTimer,\, o \in Clock : Detach(s,\, o))$

$u \triangleq \langle subject,\, observers,\, subjectTime,\, observerTime\rangle$

$Liveness \triangleq \mathrm{WF}_u(Update)$

$Spec \triangleq Init \land \Box[Next]_u \land Liveness$

THEOREM $Spec \Rightarrow \Box Invariant$

ABOUT THE AUTHORS

Toufik Taibi received his PhD from Multimedia University, Malaysia in 2003. He is currently a post-doctoral fellow at the department of Electrical and Computer Engineering at the University of Western Ontario, Canada. Prior to that he was Assistant Professor at United Arab Emirates University, UAE. Dr. Taibi has more than 16 years of combined teaching, research and software industry experience. His research interests include formal methods as it applied to software engineering, multi-agent systems and cooperative distributed systems. He can be reached at ttaibi@uwo.ca.

Ángel Herranz is an associate professor of computer science and member of the Babel Research Group at Universidad Politécnica de Madrid. He is a partner in Sistemas de Identificación Giro, an electronic engineering start-up. His research interests include formal aspects of software development, assisted and automatic program derivation and programming languages. He can be reached at aherranz@fi.upm.es.

Juan José Moreno-Navarro received his Ph.D. degree in Computer Science from the Technical University of Madrid in 1989. Since 1996 he is full professor at its Department of Computer Science, where he leads the research group Babel, focused on high-quality software development through the use of declarative technology. Currently he is also Deputy Director of IMDEA-Software (Madrid Research Institute for Software Technologies). His research lines cover all the topics related to declarative technology and software development. He can be reached at jjmoreno@fi.upm.es.