Int. J. Appl. Math. Comput. Sci., 2003, Vol. 13, No. 2, 239–253

MODELING OF DISTRIBUTED OBJECTS COMPUTING DESIGN PATTERN COMBINATIONS USING A FORMAL SPECIFICATION LANGUAGE

TOUFIK TAIBI*, DAVID CHEK LING NGO**

* Faculty of Information Technology, Multimedia University
Jalan Multimedia, 63100 Cyberjaya, Selangor, Malaysia
e-mail: toufik.taibi@mmu.edu.my

** Faculty of Information Science and Technology, Multimedia University
Jalan Ayer Keroh Lama, 75450 Melaka, Malaysia
e-mail: david.ngo@mmu.edu.my

Design patterns help us to respond to the challenges faced while developing Distributed Object Computing (DOC) applications by shifting developers' focus to high-level design concerns, rather than platform specific details. However, due to the inherent ambiguity of the existing textual and graphical descriptions of the design patterns, users are faced with difficulties in understanding when and how to use them. Since design patterns are seldom used in isolation but are usually combined to solve complex problems, the above-mentioned difficulties have even worsened. The formal specification of design patterns and their combination is not meant to replace the existing means of describing patterns, but to complement them in order to achieve accuracy and to allow rigorous reasoning about them. The main problem of the existing formal specification languages for design patterns is the lack of completeness. This is mainly because they tend to focus on specifying either the structural or behavioral aspects of design patterns but not both of them. Moreover, none of them even ventured in specifying DOC patterns and pattern combinations. We propose a simple yet Balanced Pattern Specification Language (BPSL) aimed to achieve equilibrium by specifying both the aspects of design patterns. The language combines two subsets of logic: one from the First-Order Logic (FOL) and the other from the Temporal Logic of Actions (TLA).

Keywords: Balanced Pattern Specification Language (BPSL), First-Order Logic (FOL), Temporal Logic of Actions (TLA), substitution, addition, elimination

1. Introduction

Advances in network technology have shifted the software development from stand-alone to Distributed Object Computing (DOC) in order to take advantage of their inherent benefits such as resource sharing, openness, concurrency, scalability, fault tolerance and transparency. In the remainder of this paper, distributed applications and DOC applications will be used interchangeably as Object-Oriented (OO) technology has become a standard in all kinds of software development. Despite the benefits they offer, distributed applications are harder to develop than stand-alone applications. Developers must properly address issues that are either not relevant or less problematic for stand-alone applications such as service access configuration, event handling, concurrency and synchronization (Schmidt *et al.*, 2000).

Design patterns are abstractions generated from valuable experiences of developers in solving problems repeatedly encountered within certain contexts. Design patterns and patterns are used interchangeably throughout the paper. Since patterns have been extensively tested and used in many development efforts, reusing them yields better quality software within reduced time frame. Patterns help capture and reuse the static structure and dynamic collaborations of key participants in software design. When inter-related patterns are put together, they form a "*pattern language*", which defines a vocabulary for dealing with software problems, and provide a process for orderly resolution of these problems. In recent years, it has become clear that patterns and pattern languages help alleviate problems encountered in developing distributed applications. When used properly, patterns direct developers' attention towards higher-level software applications architecture and design concerns, rather than towards low-level operating system and networking protocols and mechanisms.

Pioneer pattern writers needed an urgent means to describe these cumulated experiences in order to allow developers to reuse them. A combination of textual descriptions, OO graphical modeling languages and sample code fragments was, at that stage of pattern evolution, sufficient

for conveying the essence of patterns. However, as soon as the number of patterns had grown, and problems requiring pattern combination had surfaced, users started to realize that informal (textual and graphical) descriptions can be ambiguous, imprecise and sometimes misleading in understanding and applying patterns. Unsettled debates were initiated among users and even pattern writers themselves about various aspects of patterns (Vlissides, 1997a; 1997b). Thus, the wide dissemination of patterns in such cases depends upon the expressive ability of pattern writers. Also, an ambiguous specification can lead to incorrect usage of the pattern. Informal specifications make it difficult (or sometimes impossible) to accurately answer the following questions: Is one pattern the same as another (duplication)? Is one pattern obtained from a minor revision of another (refinement)? Are two patterns unrelated (disjointness)? These features are also critical as new patterns are being discovered, discussed and debated about.

Hence, there was a need for a formal means of accurately describing patterns. The formal specification of patterns is not meant to replace the existing informal descriptions but rather to complement them in order to achieve well-defined semantics and allow rigorous reasoning about patterns. The formal specification of patterns can help pattern users decide which pattern(s) is (are) more appropriate to solve a given design problem within a context. It can help formalize pattern combination. Finally, it can facilitate tool support for pattern usage, e.g., by finding instances of patterns in programs and by fine-tuning them to meet pattern specifications (Eden and Hirshfeld, 2001) that are stored in the so-called pattern repository. Another possibility is a tool to instantiate from a pattern specification, a possible implementation in a chosen programming language.

As the pattern field had matured, a number of formal specification languages (Chinnasamy *et al.*, 1999) have emerged as a need 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 patterns should be formalized. Patterns have a structural as well as behavioral aspect. Both the aspects are of equal importance for understanding the underlying semantics of any pattern, but the focus should always be on the predominant aspect (if any). The main problem of the available formal specification languages for patterns is the lack of completeness. This is mainly because they either were not originally conceived to specify patterns and have been adapted to do so, or they tend to focus on specifying either the structural or the behavioral aspect of patterns, but not both of them. Moreover, few of them attempted to formalize pattern combinations and none of them even ventured on specifying the DOC patterns.

We have developed a Balanced Pattern Specification Language (BPSL) (Taibi and Ngo, 2002) that is meant to accurately convey the essence of patterns in a balanced way. The BPSL was inspired by our views of why and how patterns should be formalized (Taibi and Ngo, 2001). We believe that integrating the formal specification of structural and behavioral aspects of patterns in one specification language would help formally specify patterns in a balanced way. This has led to make the BPSL develop from both First Order Logic (FOL) to specify the structural aspect, and the Temporal Logic of Actions (TLA) to specify the behavioral aspect. Moreover, since patterns are seldom used in isolation, but most of the time are combined in order to solve complex problems, we have successfully used the BPSL to formalize a pattern combination. Since the specification of the two aspects (structural and behavioral) of patterns were derived from two different logics (FOL and TLA), it is obvious that specifying a pattern combination will be done separately for each aspect. In order to specify the structural part of the combined pattern, the concept of FOL *substitutions and elimination* are applied. In order to specify the behavioral part of the combined pattern, *substitutions* (full and partial) and *additions* are applied to temporal relations and actions. All the above-mentioned concepts will be detailed in Section 3.1. We found that any significant behavioral aspect of the underlying patterns has an impact on the specification of the structural aspect of the combined pattern. This indeed shows the synergy that exists between the two complementary aspects of a pattern.

The rest of the paper is organized as follows. Section 2 gives a detailed description of the BPSL. Section 3 describes how pattern combination is formally specified using the BPSL. In Section 4 we apply the BPSL to formally specify the *Reactor* and *Leader/Followers* architectural patterns and their combination. Finally, in Section 5 we present work related to what is presented in this paper, and we conclude the paper in Section 6.

2. Balanced Pattern Specification Language (BPSL)

Patterns differ in terms of their field of usage, the problem they solve and its context. However, each pattern has a structural aspect and a behavioral aspect. In certain patterns the structural aspect is predominant while in others the behavioral aspect is supreme. Hence, any formal specification that is claimed to completely describe patterns should incorporate the specification of both the structural and behavior aspects (Taibi and Ngo, 2001). Each pattern can be seen from two complementary views: the structural view and the behavioral view. However, the predominant view should always be the focus. This principle is not new, but comes directly from the very nature of OO

systems. By a balance in BPSL we mean that the formal specification of both the structural and behavioral aspects of patterns should complement each other. As the BPSL is aimed at describing 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. To understand that, the predominant aspect can be made the focus. Once this is achieved, users will be able to know when and how to use a given pattern, which is crucial to take full advantage of the inherent benefits of patterns.

The structural aspect of a pattern can be formalized using a subset of First Order Logic (FOL), because the relations between pattern participants can be easily expressed as predicates. For simplicity, the subset of the FOL used focuses on variable and predicate symbols. The behavioral aspect of a pattern can be formalized using a subset of Temporal Logic of Actions (TLA) (Lamport, 1994), 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 objects through temporal relations. The following are the building blocks of the BPSL. They reflect entities (participants) and relations (collaborations) between them in a 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 2.2).
2. Relations express the way in which entities collaborate. They can be either permanent or temporal. Once defined, permanent relations between entities cannot be changed while temporal relations may change throughout the execution of actions. The BPSL defines a set of primary permanent relations based on what 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 patterns. Actions associate and disassociate objects through temporal relations.
4. Any newly defined entity or permanent relation must by derived from primary entities and *primary* permanent relations, respectively.

2.1. Structural Aspect Specification

The subset of FOL used to describe the structural aspect of patterns comprises variable symbols, connectives (mainly '$\wedge$'), quantifiers (mainly '$\exists$') and predicate symbols acting upon variable symbols. The variable symbols

Table 1. Primary permanent relations and their intent.

Name	Domain	Intent
Defined-in	$\boldsymbol{M} \times \boldsymbol{C}$	Indicates that a method is defined in a certain class.
	$\boldsymbol{A} \times \boldsymbol{C}$	Indicates that an attribute is defined in a certain class.
Reference-to-one (-many)	$\boldsymbol{C} \times \boldsymbol{C}$	Indicates that one class defines a member whose type is a reference to one (many) instance(s) of the second class.
Inheritance	$\boldsymbol{C} \times \boldsymbol{C}$	Indicated that the first class inherits from the second.
Creation	$\boldsymbol{M} \times \boldsymbol{C}$	Indicates that a method contains an instruction that creates a new instance of a class.
	$\boldsymbol{C} \times \boldsymbol{C}$	Indicates that one of the methods of a class contains an instruction that creates a new instance of another class.
Invocation	$\boldsymbol{M} \times \boldsymbol{M}$	Indicates that the first method invokes the second method.
	$\boldsymbol{C} \times \boldsymbol{M}$	Indicates that a method of a class invokes a specific method of another class.
	$\boldsymbol{M} \times \boldsymbol{C}$	Indicates that a specific method of a class invokes a method of another class.
	$\boldsymbol{C} \times \boldsymbol{C}$	Indicates that a method of a class invokes a method of another class.
Argument	$\boldsymbol{C} \times \boldsymbol{M}$	Indicates that a reference to a class is an argument of a method.
	$\boldsymbol{A} \times \boldsymbol{M}$	Indicates that an attribute is an argument of a method.
	$\boldsymbol{V} \times \boldsymbol{M}$	Indicates that an untyped value is an argument of a method.
Return-type	$\boldsymbol{C} \times \boldsymbol{M}$	Indicates that a method returns a reference to a class.
	$\boldsymbol{O} \times \boldsymbol{M}$	Indicates that a method returns an object.
Instance	$\boldsymbol{O} \times \boldsymbol{C}$	Indicates that an object is an instance of a certain class.

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 denoted respectively by $\boldsymbol{C}$, $\boldsymbol{A}$, $\boldsymbol{M}$, $\boldsymbol{O}$, and $\boldsymbol{V}$. Table 1 depicts the *primary* permanent relations, their domain and their intent. These relations straightforwardly come from object-oriented technology concepts. It is the smallest set (in terms of the 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* $(a_1, m_1) \wedge \cdots \wedge$ *Argument* $(a_n, m_1) \wedge$ *Argument* $(a_1, m_2) \wedge \cdots \wedge$ *Argument* (a_n, m_2), where $m_1, m_2 \in M$ and $a_1, \ldots, a_n \in C \cup A \cup V$, which means that they can either be references to classes, attributes or untyped values. *Primary* permanent relations are general in the sense that they can be used to specify all patterns. Primary permanent relations can be easily extracted from the structure of the patterns represented usually by a Unified Modeling Language (UML) (Rambaugh *et al.*, 1998) class diagram and the collaboration of the pattern participants represented by UML sequence diagrams.

2.2. Behavioral Aspect Specification

For patterns that have a predominant behavioral aspect, it is necessary to understand how objects collaborate to achieve the expected behavior. The subset of TLA used in the BPSL looks at a pattern as an action system (Back and Kurki-Suonio, 1988) that can be regarded as an abstract state machine consisting of state variables (class attributes) and a set of actions, where each action consists of a precondition and a body. Actions change state variables and/or associate and disassociate objects through temporal relations.

TLA deals with behaviors $\sigma = (S_0, S_1, \ldots)$ defined as an infinite sequence of states. Each state S_i is a collection of values of state variables. A pair of consecutive states (S_i, S_{i+1}) in a behavior is called a transition. A stuttering transition is the one in which the state variables do not change from S_i to S_{i+1}.

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

TR$(o1, o2)$ means that an object $o1$ of a class $C1$ is currently associated through *TR* with an object $o2$ of a class $C2$, while $\neg TR\,(o1, o2)$ signifies that $o1$ and $o2$ are no longer associated through *TR*. *TR*$(o1, C2)$ indicates that $o1$ is associated with all instances (objects) of the 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 or state change caused by the execution of the action. For example, if we suppose that a class C has x as an attribute, an action A may be defined as follows: $A(o, p) : o.x \neq p \rightarrow o.x' = p$, where o is an object of the class C, and p denotes an untyped value. The symbol ':' means 'by definition'. The expression $o.x \neq p$ is the precondition under which the action can be executed and $o.x' = p$ 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. They define a state change caused by the execution of the action. Objects that participate in an action and untyped values are non-deterministically selected from those that are suitable. For example, the above action is allowed for all objects having $o.x \neq p$. Semantically, an action is a Boolean expression that is true or false with regard to a pair of states, with primed variables referring to the second state. 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 the value that state T assigns to x is equal to p.

Unlike permanent relations, temporal relations and actions are specific to each pattern and therefore have to be defined separately for each pattern. The system starts in some initial state. As time elapses, actions are executed, changing the system 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 has been started, it cannot be interrupted or interfered by other actions. The computational model is *interleaving*, that is, only one action at a time is being executed. The interleaving model is not a model of execution but a model of observation so as to facilitate the reasoning about the behavioral aspect of the pattern.

The sequence of states describing the execution of an action is potentially infinite. The properties of a system 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 indefinitely, i.e. there are no infinite stuttering transitions. Safety can be guaranteed by ensuring that invariants are true at all states of the system while liveliness is obtained

by imposing an explicit fairness requirement. Marking an object with an asterisk '*' denotes a fairness requirement, stating that if an object can repeatedly participate in an action, the action will be executed for the object.

2.3. Integrating the Structural and Behavioral Aspect Specifications

A formula in the BPSL has the following form:

$$\exists(x_1,\ldots,x_n):\begin{cases}\wedge_i PR_i(a_i,b_i) & \text{(Permanent relations)},\\ \wedge j\, TR_j(c_j,d_j) & \text{(Temporal relations)},\\ \vee_k A_k(\ldots) & \text{(Actions)},\end{cases}$$

where the PR_i's are permanent relation symbols, the TR_j's are temporal relation symbols and the A_k's are action symbols while $x_1,\ldots,x_n$ 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 in Table 1, while in the notation $TR_j(c_j,d_j)$, c_j and d_j represent classes. The notation $A_k(\ldots)$ means that actions can have any number of arguments that should be either objects or untyped values. Any argument of PR_i, TR_j and A_k is a subset of $\{x_1,\ldots,x_n\}$. The variables $x_1,\ldots,x_n$ are typed, and each represents an entity, as expected (see Section 2.1). In all specifications, we follow a convention in which only relations and actions start with a capital letter. The BPSL does not use two disjoint subsets of variables, whereby the variables of the first subset participate in permanent relations and those of the second participate in temporal relations and actions. Indeed, the variables $x_1,\ldots,x_n$ participate in permanent and temporal relations, as well as actions. This proves that a seamless combination of the two aspects (structural and behavioral) was implemented in the 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 design pattern. The first declares 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 kinds of possible behavior (an infinite sequence of states), as seen in Section 2.2.

3. Formally Specifying a Pattern Combination

Component based software engineering focuses on building software systems by assembling previously developed and well-tested or even formally validated components (D'Souza and Wills, 1998), rather than developing the entire system from scratch. This leads to an apparent reduction in the cost and effort. Moreover, it can help to reduce the challenges faced while developing distributed applications. However, the assembly can lead to software failures if it is not done with prior knowledge of the salient properties and invariants of each component (Cheesman and Daniels, 2000). Patterns are special types of components offering a better understanding of the design assumptions, trade-offs and implications of a component's implementation. Since each pattern represents a well-tested abstraction that has an infinite number of instances (implementations), patterns are considered as building blocks from which more reusable and changeable software designs can be built. Thus, if formalized, pattern combinations can lead to ready-made architectures from which only instantiation is required to build robust implementations.

In the previous section we were able to develop a formal language to specify patterns. While this is an important milestone in achieving a better understanding and successful usage of patterns, it is still considered insufficient, as patterns are seldom used in isolation. While developing software, in most cases two or more patterns are to be combined to solve a given problem within a context. In this sense patterns represent micro-architectures that, when glued together, form the whole software architecture. Hence, there is a need to achieve a second milestone by formally specifying a pattern combination which is harder to understand and use than individual patterns.

The *completeness* of a component combination (sometimes called the composition) is usually defined by two conditions (Dong *et al.*, 2000):

(i) no component loses any properties after combination,

(ii) no new properties about each component can be inferred from the combination.

Any combination that satisfies both of the conditions is said to be *faithful* or *correct*. While the second condition is also applicable to patterns, we will soon see that the first condition is not always applicable to patterns. Since the specification of the two aspects (structural and behavioral) of patterns were derived from two different kinds of logic (FOL and TLA), it is obvious that specifying a pattern combination will be done independently for each aspect. In order to achieve good readability of formulae, in the reminder of this section we assume the combination of only two patterns. However, the same results can be applied to any number of patterns.

3.1. Pattern Combination Constructs

In order to specify the structural part of the combined pattern, the concept of FOL *substitutions* and *elimination* are applied. In order to specify the behavioral part of the combined pattern, *substitutions* (full and partial) and *additions* are applied to temporal relations and actions. In FOL, a *substitution* list $\Theta = \{v_1/t_1, \ldots, v_n/t_n\}$ means to replace all occurrences of the variable symbol v_i by the terms t_i. Substitutions are made from left to right in the list. For example, *subst*($\{x$/*Pasta*,y/*John*$\}$,*eats*$(y, x)) =$ *eats* (*John*, *Pasta*). In the BPSL we restrict the terms t_i to variable symbols only, i.e., constant and function symbols are not supported, as they are not used in the subset of FOL used by the BPSL. Substitutions can also be defined on temporal relations and actions. For example, $TR2(x, y) = subst(\{a/x, b/y\}, TR1(a, b))$ and $A2(x, y) = subst(\{a/x, b/y\}, A1(a, b))$. *Elimination* is defined as taking out certain entities (variables) and permanent relations (predicates) from the formula that specifies the structural aspect of a pattern. Obviously, the elimination of a variable symbol triggers the automatic elimination of all predicates that involve that variable symbol. For example, $elim(\{x,\ single(y)\}, \exists x, y\ smart(x) \wedge likes(x, y) \wedge student(y) \wedge single(y)) = \exists y\ student(y)$. *Addition* is defined as adding object variables to actions. For example, if an action $A1$ is defined as $A1(a, b) : TR1(a, b) \wedge TR2(a, b) \rightarrow TR3'(a, b)$, we can define a new action $A2$ based on $A1$ as follows: $A2(x, a, b) : add(x, subst(\{a/x\ in\ TR1\}, A1))$ which yields $A2(x, a, b) : TR1(x, b) \wedge TR2(a, b) \rightarrow TR3'(a, b).subst\{a/x\ in\ TR1\}$ means that the substitution is *partial*, i.e., we only replace a by x in the temporal relation *TR*1, which is part of the precondition of action $A1$. We can also define a new action $A3$ based on $A2$ by adding a precondition as follows: $A3(x, a, b) : A2(x, a, b) \wedge TR4(x, a)$ which yields $A3(x, a, b) : TR1(x, b) \wedge TR2(a, b) \wedge TR4(x, a) \rightarrow TR3'(a, b)$.

3.2. BPSL's Combination Process

Table 2 depicts the BPSL's combination process that uses all the techniques defined in the previous subsection (substitution, partial substitution, elimination, addition and precondition addition) and was the result of many experiments on combining different patterns from (Gamma *et al.*, 1995; Schmidt *et al.*, 2000). We will apply most of the findings of this section in the case study of Section 4.3. The concept of *behavioral dominance*, i.e., which pattern behaviorally dominates the other, is crucial when combining patterns that have a significant behavioral aspect. Behavioral dominance is case-dependent and can only be derived after studying the behavior of the underlying patterns and the expected behavior of their combination.

The specification of the combination for both parts (structural and behavioral) is guaranteed to be correct because it is built on top of the existing well-tested specifications of the patterns involved in the combination. As such, instances of the pattern combination can be used in particular applications without further proof, which saves considerable efforts of fixing errors downstream in the software development process. Consistency checking of a specification of a combined pattern is simply based on the fact that the specification of the combined pattern follows the process and formulae defined in Table 2.

4. Case Study: Reactor-Leader/Followers Pattern Combination

In the following three subsections, we will show how BPSL was successfully used to formally specify *Reactor* and *Leader/Followers* architectural patterns (which have a good mixture of the structural and behavioral aspects) as well as their combination. In (Schmidt *et al.*, 2000), the *Reactor* pattern was classified as an *event handling* pattern while the *Leader/Followers* pattern as a *concurrency* pattern.

4.1. Reactor Architectural Pattern

The *Reactor* architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

The *Reactor* architectural pattern synchronously waits for the arrival of indication events on one or more event sources, such as connected socket handles. It integrates the mechanisms that demultiplex and dispatch the events to services that process them. It decouples event demultiplexing and dispatching mechanisms from the application-specific processing of indication events within the services. The *Reactor* architectural pattern is applied to event-driven applications that receive requests simultaneously, but process them synchronously and serially.

Figure 1 depicts the UML class diagram of the *Reactor* pattern. For each service an application offers, the pattern introduces a separate *event handler* that process certain types of events from certain event sources. *Event handlers* register with the *reactor*, which uses a *synchronous event demultiplexer* (such as *select*( ) in UNIX) to wait for indication events to occur on one or more event sources. When indication events occur, the *synchronous event demultiplexer* notifies the *reactor*, which then synchronously dispatches the *event handler* associated with the event so that it can perform the requested service.

Table 2. BPSL's combination process.

Declarations

Let $P1$ and $P2$ be the patterns to be combined
P be the combined pattern
φ be the formula that specifies the structural aspect of P
$\varphi 1$ be the formula that specifies the structural aspect of $P1$
$\varphi 2$ be the formula that specifies the structural aspect of $P2$
$\varphi 3$ be extra entities and/or permanent relations that may be needed after combining $P1$ and $P2$
$TR1$ be a temporal relation of $P1$
$TR2$ be a temporal relation of $P2$
$TR3$ be a temporal relation of P
$A1$ be an action of $P1$
$A2$ be an action of $P2$
A be an action of P
PC be a precondition representing an extra temporal relation (from $P1$ or $P2$) added to one of P's actions
$w_1, \ldots, w_n$ variables of $P1$ (or $P2$)
$v_1, \ldots, v_n$ variables of $P1$ (or $P2$)
$u_1, \ldots, u_m$ variables of $P1$ (or $P2$)
bdom be an operator that returns the formula, temporal relation or action of the behaviorally dominant pattern

Algorithm

If $P1$ and $P2$ have no significant behavioral aspect, then $\varphi = subst(\{v_1/w_1, \ldots, v_n/w_n\}, \varphi 1 \wedge \varphi 2)$ (1)
else if $P1$ behaviorally dominates $P2$ (or vice versa) then

$$\begin{cases} \varphi = subst(\{v_1/w_1, \ldots, v_n/w_n\}, bdom(\varphi 1, \varphi 2)) \wedge \varphi 3, & (2) \\ TR3(w_1, \ldots, w_n) = subst(\{v_1/w_1, \ldots, v_n/w_n\}, bdom(TR1, TR2)(v_1, \ldots, v_n)), & (3) \\ A3(w_1, \ldots, w_n) = subst(\{v_1/w_1, \ldots, v_n/w_n\}, bdom(A1, A2)(v_1, \ldots, v_n)), & (4) \end{cases}$$

else

$$\begin{cases} \varphi = elim(\{\ldots\}, \varphi 1 \wedge \varphi 2) \wedge \varphi 3, & (5) \\ TR3(\ldots) = TR1(TR2)(\ldots) \text{ or}, & (6) \\ TR3(w_1, \ldots, w_n) = subst(\{v_1/w_1, \ldots, v_n/w_n\}, TR1(TR2)(v_1, \ldots, v_n)), & (7) \\ A3(\ldots) : A1(A2)(\ldots), \text{ or} & (8) \\ A3(w_1, \ldots, w_n) : subst(\{v_1/w_1, \ldots, v_n/w_n\}, A1(A2)(v_1, \ldots, v_n)), \text{ or} & (9) \\ A3(u_1, \ldots, u_m, v_1, \ldots, v_n) : add(u_1, \ldots, u_m, A1(A2)(v_1, \ldots, v_n)) \wedge [PC], \text{ or} & (10) \\ \text{any combination of (9) and (10).} & (11) \end{cases}$$

Notes

*TR*1(*TR*2) means either *TR*1 or *TR*2.
[ ] means optional.
In (1) and (2) the variables v_i and w_i represent classes, attributes or methods.
In (3) and (7) the variables v_i and w_I represent classes.
In (4), (9) and (10) the variables u_i, v_i, and w_i represent objects.
(2) and (5) show that any significant behavioral aspect of the underlying patterns has an impact on the specification of the structural aspect of the combined pattern. This indeed shows the synergy that exists between the two complementary aspects of a pattern.
(5) shows indeed that after combination certain properties of the underlying patterns might be lost because of the *elim* operation.
In (3) and (4), in some cases, extra temporal relations and actions might be added to accommodate the behavior required by the behaviorally non-dominant pattern.
In (6)–(11), in some cases, extra temporal relations and actions might be added to accommodate the behavior of the combined pattern.
In (9) and (11) substitution may be partial.

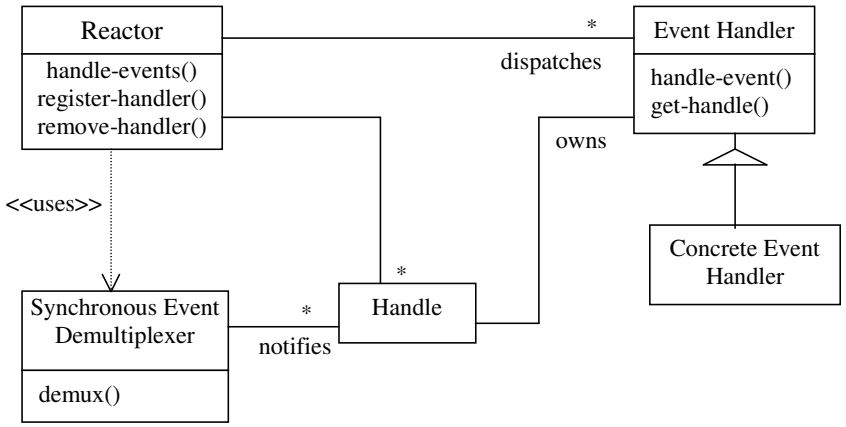

Fig. 1. UML class diagram of the *Reactor* architectural pattern.

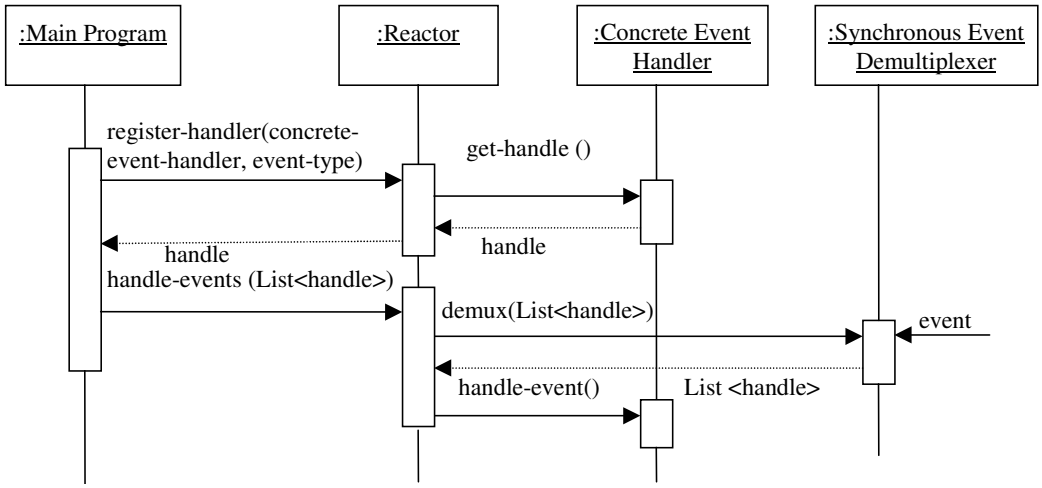

Fig. 2. UML sequence diagram of the *Reactor* architectural pattern.

Figure 2 depicts the UML sequence diagram of the *Reactor* pattern. Firstly, an application registers a specific *event handler* with the *reactor* by indicating the type of indication events(s) the *event handler* wants the *reactor* to notify it about, when such event(s) occur on the associated *handle*. The keyword *List*<> is used to reflect a list of objects of the class in angle brackets. Secondly, the *reactor* instructs each concrete *event handler* to provide its internal *handle* by invoking the *get-handle*( ) method. The *handle* identifies the source of indication events to the *synchronous event demultiplexer* and the operating system. Thirdly, the application starts the *reactor*'s event loop *handle-events*( ). At this point the *reactor* combines the handles from each registered concrete event hander into a *handle set*. It then calls the method *demux*( ) of the *synchronous event demultiplexer* to wait for indication events to occur on the *handle set*. The *synchronous event demultiplexer* method *demux*( ) (*select*( ) in the case of UNIX) returns to the *reactor* when one or more handles corresponding to event sources become ready (for example a socket becomes ready to read). Finally, the *reactor* uses the ready handles as '*keys*' to locate the appropriate *event handler*(s) and dispatch *handle-event*( ) method.

Table 3 depicts the BPSL specification of the *Reactor* pattern. It starts by defining the entities in the pattern, which are limited in this case to classes, methods, an untyped-value and objects. The second compartment of Table 2 depicts the permanent relations, most of which can be derived from Figs. 1 and 2. The permanent relation *Instance* indicates that a given object is an instance of a given class. Objects are used in the specification of actions. The third compartment of Table 3 depicts temporal relations. In the case of the *Reactor* pattern, before a *concrete-event-handle* can start processing responses to detected events, it first has to register with the *reactor* yielding the temporal relation *Registered*. Each *concrete-event-handle* owns a *handle* that is used to wait for events to occur on it, yielding the temporal relation *Owned*.

Table 3. BPSL specification of the *Reactor* architectural pattern.

$\exists reactor, event-handler, concrete-event-handler, handle, synchronous-event-demultiplexer \in C;$ $handle-events, register-handler, remove-handler, handle-event, get-handle, demux \in M;$ $event-type \in V;$ $r, h, ceh, sed \in O;$
Defined-in (*handle-events, reactor*) $\wedge$ *Defined-in* (*register-handler, reactor*) $\wedge$ *Defined-in* (*remove-handler, reactor*) $\wedge$ *Defined-in* (*handle-event, event-handler*) $\wedge$ *Defined-in* (*get-handle, event-handler*) $\wedge$ *Defined-in* (*demux, synchronous-event-demultiplexer*) $\wedge$ *Reference-to-many* (*reactor, event-handler*) $\wedge$ *Reference-to-many* (*reactor, handle*) $\wedge$ *Reference-to-many* (*synchronous-event-demultiplexer, handle*) $\wedge$ *Reference-to-one* (*event-handler, handle*) $\wedge$ *Reference-to-one* (*reactor, synchronous-event-demultiplexer*) $\wedge$ *Inheritance* (*concrete-event-handler, event-handler*) $\wedge$ *Invocation* (*register-handler, get-handle*) $\wedge$ *Invocation* (*handle-events, demux*) $\wedge$ *Invocation* (*handle-events, handle-event*) $\wedge$ *Argument* (*concrete-event-handler, register-handler*) *Argument* (*event-type, register-handler*) *Argument* (*concrete- event-handler, remove-handler*) $\wedge$ *Argument* (*handle, handle-events*) $\wedge$ *Argument* (*handle, demux*) $\wedge$ *Return-type* (*handle, get-handle*) $\wedge$ *Return-type* (*handle, demux*) *Instance* (*r, reactor*) $\wedge$ *Instance* (*h, handle*) $\wedge$ *Instance* (*ceh, concrete-event-handler*) $\wedge$ *Instance* (*sed, syschronous-event-demultiplexer*)
Registered (*reactor* $[0..1]$,*concrete-event-handler* $[*]$) $\wedge$ *Owned* (*handle* $[0..1]$, *concrete-event-handler* $[0..1]$) $\wedge$ *Ready* (*handle* $[*]$, *synchronous-event-demultiplexer* $[0..1]$) $\wedge$ *Dispatched* (*reactor* $[0..1]$, *concrete-event-handler* $[*]$)
Register (*r,ceh,h*)*:* $\neg$ *Registered* (*r,ceh*) $\wedge \neg$ *Owned* (*handle, ceh*) $\rightarrow$ *Registered'* (*r,ceh*) $\wedge$ *Owned'* (*h,ceh*) $\vee$ *Remove* (*r,ceh,h*)*: Registered* (*r,ceh*) $\wedge$ *Owned* (*h, ceh*) $\wedge$ *Dispatched* (*r,ceh*) $\rightarrow \neg$ *Registered'* (*r,ceh*) $\wedge \neg$ *Owned'* (*h,ceh*) $\vee$ *Dispatch* (*sed,r,ceh,h*)*: Ready* (*h,sed*) $\wedge$ *Owned* (*h,ceh*) $\wedge$ *Registered* (*r,ceh*) $\rightarrow$ *Dispatched'* (*r,ceh*)

The *synchronous-event-demultiplexer* knows when handles are ready if the type of events they handle has occurred, yielding the temporal relation *Ready*. When a *handle* of a *concrete-event-handler* becomes ready, the *reactor* dispatches this information to the *concrete-event-handler* that owns it in order to do the required processing, yielding the temporal relation *Dispatched*. The fourth and last compartment of Table 3 depicts actions of the *Reactor* pattern. Action *Register* reflects that for a *concrete-event-handler* to register with the *reactor*, it should not be already registered with the *reactor* and not owning any *handle* to be later registered and owning a given *handle*. Action *Remove* reflects that for a *concrete-event-handler* to be removed from the *reactor* it must be registered and owns a *handle* and the *reactor* has already dispatched to it the last event that occurred on the *handle* that it owns. When the action is executed, the *concrete-event-handler* will no longer be registered with the *reactor* and will not own the *handle* that was allocated to it. The action *Dispatch* reflects that for the *reactor* to dispatch an event to a *concrete-event-handler*, the latter must be registered with the *reactor*, and *owns* a *handle*, which is ready.

4.2. The Leader/Followers Architectural Pattern

The *Leader/Followers* architectural pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch and process service requests that occur on the event sources. The *Leader/Followers* pattern does the above by structuring a pool of threads to share a set of event sources efficiently by taking turns demultiplexing events that arrive on these event sources and synchronously dispatching the events to application services that process them. The pattern is applied to event-driven applications where multiple service requests arriving on a set of event sources must be processed efficiently by multiple threads that share the event sources. A *thread pool* is a group of threads that share a synchronizer such as a semaphore or condition variable, and implement a protocol for coordinating their transition between various roles. One or more threads play the follower role and queue up on the thread pool synchronizer waiting to play the leader role. One of these threads is selected to be the leader, which waits for an event to occur on any *handle* in its *handle set*. When an event occurs, the current leader thread promotes a follower thread to become the new leader. The original leader then concurrently plays the role of a processing thread, which demultiplexes that event from the *handle set* to an appropriate *concrete-event-handler* and dispatches the *handle-event*( ) method. A processing thread can execute concurrently with the leader thread and all other threads that are in the processing state. After a processing thread has finished handling an event, it returns to playing the role of a follower thread and waits on the *thread pool* synchronizer for its turn to become the leader thread again.

The *Reactor* pattern often forms the core of the *Leader/Followers* pattern implementations. However, the *Reactor* pattern can be used instead of the *Leader/Followers* when each event only requires a short amount of time to process. In this case the additional scheduling complexity of the *Leader/Followers* pattern is unnecessary.

Table 4 depicts the BPSL specification of the *Leader/Followers* pattern. It starts by defining the entities in the pattern, which are limited in this case to classes, an attribute, methods and objects (which will be used in the specification of the behavioral aspect). The second compartment of Table 4 depicts the permanent relations, most of which can be straightforwardly derived from the class diagram of Fig. 3. The third compartment of Table 4 depicts temporal relations. The temporal relation *Owned* is identical to the one defined in the *Reactor* pattern. The temporal relations *Ready* and *Dispatched* have the same semantics as in the *Reactor* pattern with the exception that the class *handle-set* replaces the class *synchronous-event-demultiplexer* in *Ready* and the class *thread-pool* replaces the class *reactor* in *Dispatched*. The temporal relation *Activated* reflects whether or not a *handle* associated with a *concrete-event-handler* is activated. As explained earlier, only a leader thread is allowed to synchronously demultiplex events on handles. Once an event occurs on a *handle*, a new thread (from the *thread pool*) is promoted to the leader and the ready *handle* is deactivated. This *handle* will be reactivated only after the previous leader thread has dispatched the *handle-event*( ) method of the *concrete-event-handler*. The temporal relations *Leader, First-Follower* and *Follower(thread-pool*[*]) reflect who is the leader, first-follower and follower (respectively) in the *thread pool*.

The last compartment of Table 4 depicts actions of the *Leader/Followers* pattern. The action *Promote* reflects that in order to promote a thread (*tp*2) to become the leader the following is required:

(i) (*tp*2) must be the first follower of a leader thread (*tp*1),

(ii) the *handle* (h) owned by the *concrete-event-handler* (*ceh*) must be activated,

(iii) the *handle* (h) must be ready,

(iv) the event that occurred at (h) is not yet dispatched by (*tp*1) to (*ceh*).

The execution of the action results in the following:

(i) (*tp*2) becomes the leader thread, and

(ii) the *handle* (h) becomes non-activated.

The action *Dispatch* is similar to the one defined in the *Reactor* pattern. However, here, in order to dispatch an event to its *concrete-event-handler*, the *handler* must be ready, owned by a *concrete-event-handler* and not activated. After executing the action the event becomes dispatched by the leader thread to the *concrete-event-handler* and the *handle* activated again to be able to receive more events. The last three actions are related to threads joining or rejoining the *thread pool*. In general, to join a *thread pool*, a thread should have already dispatched the event to the related *concrete-event-handler* and its associated handle is activated. The action *Join*1 reflects the case where one thread wants to join the *thread pool*, which does not have a leader and thus becomes the leader. The action *Join*2 reflects the case where a thread (*tp*2) wants to join the *thread pool*, which has a leader (*tp*1) but does not have a first follower and thus becomes the first follower itself. The action *Join*3 reflects the case where a thread (*tp*3) wants to join the *thread pool*, which has a leader (*tp*1) and a first follower (*tp*2) and thus becomes a follower.

Table 4. BPSL specification of the *Leader/Followers* architectural pattern.

$\exists$ *thread-pool, handle-set, handle, event-handler, concrete-event-handler* $\in C$*;* *synchronizer* $\in A$*;* *deactivate-handle, reactivate-handle, demux, join, promote-new-leader, handle-event, get-handle* $\in M$*;* *tp*1,*tp*2, *tp*3, *hs, h, ceh* $\in O$*;*
Defined-in (*synchronizer, thread-pool*) $\wedge$ *Defined-in* (*join, thread-pool*) $\wedge$ *Defined-in* (*promote-new-leader, thread-pool*) $\wedge$ *Defined-in* (*deactivate-handle, handle-set*) $\wedge$ *Defined-in* (*reactivate-handle, handle-set*) $\wedge$ *Defined-in* (*demux, handle-set*) $\wedge$ *Defined-in* (*handle-event, event-handler*) $\wedge$ *Defined-in* (*get-handle, event-handler*) $\wedge$ *Reference-to-one* (*thread-pool, handle-set*) $\wedge$ *Reference-to-many* (*thread-pool, event-handler*) $\wedge$ *Reference-to-many* (*handle-set, handle*) $\wedge$ *Reference-to-one* (*event-handler, handle*) $\wedge$ *Inheritance* (*concrete-event-handler, event-handler*) $\wedge$ *Invocation* (*demux, handle-event*) $\wedge$ *Invocation* (*handle-event, deactivate-handle*) $\wedge$ *Invocation* (*handle-event, promote-new-leader*) $\wedge$ *Invocation* (*join, handle-events*) $\wedge$ *Argument* (*handle, deactivate-handle*) $\wedge$ *Argument* (*handle, reactivate-handle*) $\wedge$ *Argument* (*handle, demux*) $\wedge$ *Return-type* (*handle, get-handle*) $\wedge$ *Return-type* (*handle, demux*) $\wedge$ *Instance* (*tp*1, *thread-pool*) $\wedge$ *Instance* (*tp*2,*thread-pool*) $\wedge$ *Instance* (*tp*3,*thread-pool*) $\wedge$ *Instance* (*hs, handle-set*) $\wedge$ *Instance* (*h, handle*) $\wedge$ *Instance* (*ceh, concrete-event-handler*)
Owned (*handle* [0..1], *concrete-event-handler* [0..1]) $\wedge$ *Ready* (*handle* [$*$], *handle-set* [0..1]) *Dispatched* (*thread-pool* [$*$],*concrete-event-handler* [$*$]) $\wedge$ *Activated* (*handle* [0..1],*concrete-event-handler* [0..1]) $\wedge$ *Leader* (*thread-pool* [0..1]) $\wedge$ *First-Follower* (*thread-pool* [0..1]) $\wedge$ *Follower* (*thread-pool* [$*$])
Promote (*tp*1,*tp*2,*ceh,h,hs*)*: Leader* (*tp*1) $\wedge$ *First-Follower* (*tp*2) $\wedge$ *Activated* (*h,ceh*) $\wedge$ *Ready* (*h,hs*) $\wedge \neg$ *Dispatched* (*tp*1,*ceh*) $\rightarrow$ *Leader'* (*tp*2) $\wedge \neg$ *Activated'* (*h,ceh*) $\vee$ *Dispatch*(*hs,tp,ceh,h*)*: Ready* (*h,hs*) $\wedge$ *Owned* (*h,ceh*) $\wedge \neg$ *Activated* (*h,ceh*) $\rightarrow$ *Dispatched'* (*tp,ceh*) $\wedge$ *Activated'* (*h,ceh*) $\vee$ *Join*1 (*tp*1,*h,ceh*)*: Activated* (*h,ceh*) $\wedge$ *Dispatched* (*tp*1,*ceh*) $\wedge \neq$ *Leader* (*thread-pool*) $\rightarrow$ *Leader'* (*tp*1) $\vee$ *Join*2 (*tp*1,*tp*2,*h,ceh*)*: Activated* (*h,ceh*) $\wedge$ *Dispatched* (*tp*2,*ceh*) $\wedge$ *Leader* (*tp*1) $\wedge \neg$ *First-Follower* (*thread-pool*) $\rightarrow$ *First-Follower'* (*tp*2) $\vee$ *Join*3 (*tp*1,*tp*2,*tp*3,*h,ceh*)*: Activated* (*h,ceh*) $\wedge$ *Dispatched* (*tp*3,*ceh*) $\wedge$ *Leader* (*tp*1) $\wedge$ *First-Follower* (*tp*2) $\rightarrow$ *Follower'*(*tp*3)

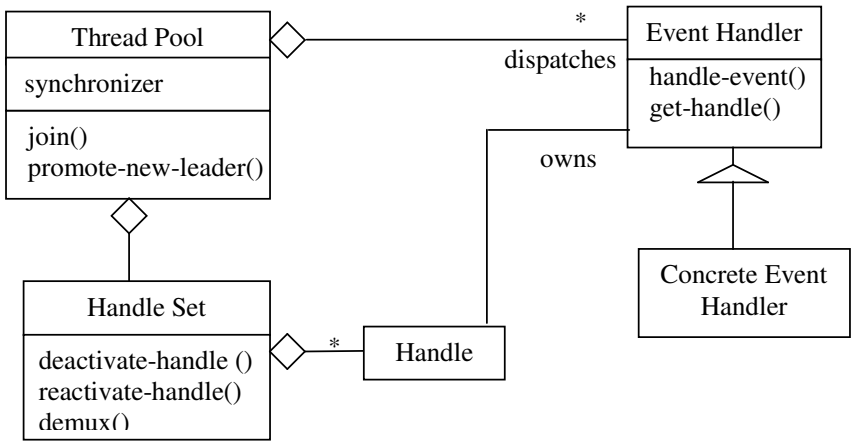

Fig. 3. UML class diagram of the *Leader/Followers* architectural pattern.

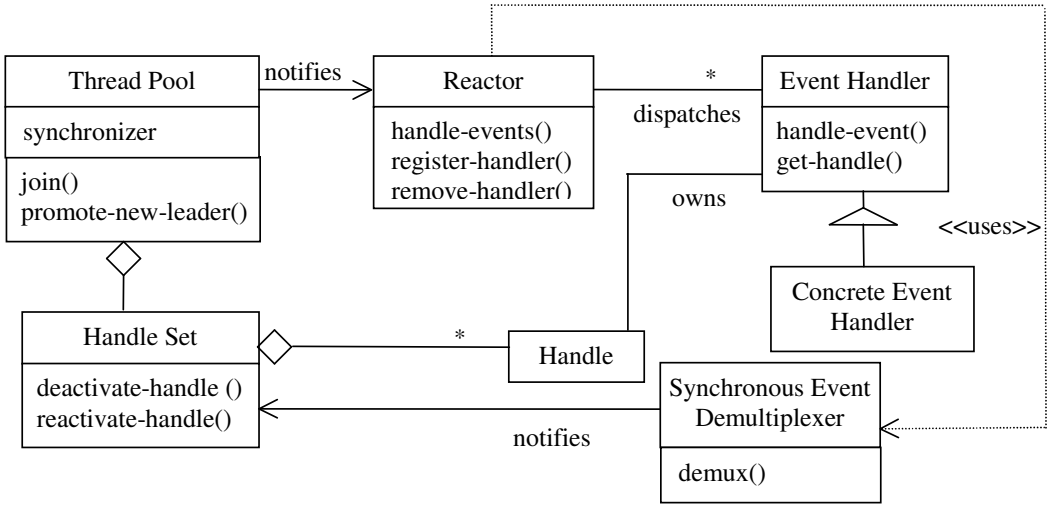

Fig. 4. UML class diagram of the *Reactor-Leader/Followers* pattern combination.

4.3. Reactor-Leader/Followers Pattern Combination

Figure 4 depicts the UML class diagram of the *Reactor-Leader/Followers* pattern combination while Table 5 depicts its BPSL specification.

Since the *Reactor* pattern is simply used as the underlying *synchronous event demultiplexer* for the *Leader/Followers* pattern, Fig. 4 really merges the diagrams of Figs. 1 and 3 with few changes. The eliminations made are as follows:

(i) The *thread pool* class is not linked to the *event handler* class.

(ii) The *reactor* class is not linked to the *handle* class.

(iii) *demux* is not defined in *handle-set*.

(iv) The *synchronous event demultiplexer* class is not linked to the *handle* class.

(v) *demux* does not invoke *handle-event*.

(vi) The object *hs* and (automatically) the *Intance* (*hs*, *handle-set*) permanent relation is not needed as the *synchronous event demuliplexer* class takes care of demultiplexing events.

The additions made are as follows:

(i) The *thread-pool* class is linked to the *reactor* class.

(ii) The *synchronous-event-demultiplexer* class is linked to the *handle-set* class.

Assume that $\varphi 1$ represents the formula that specifies the structural aspect of the *Reactor* pattern and $\varphi 2$ represents the formula that specifies the structural aspect of the *Leader/Followers* pattern. Based on the above-mentioned eliminations and additions, the formula φ that specifies the structural aspect of the combined pattern can be defined as in Compartment 1 of Table 5. The temporal

Table 5. BPSL specification of the *Reactor-Leader/Followers* pattern combination.

φ =*elim* ({*Reference-to-many* (*thread-pool, event-handler*), *Reference-to-many* (*reactor, handle*), *Defined-in* (*demux, handle-set*), *Reference-to-many* (*synchronous-event-demultiplexer, handle*), *Invocation* (*demux, handle-event*), *hs*}, $\varphi 1 \wedge \varphi 2$) $\wedge$
Reference-to-one (*thread-pool, reactor*) $\wedge$
Reference-to-one (*synchronous-event-demultiplexer, handle-set*)
Registered (*reactor* [0..1],*concrete-event-handler*[$*$]) $\wedge$
Owned (*handle* [0..1], *concrete-event-handler* [0..1]) $\wedge$
Ready (*handle* [$*$], *synchronous-event-demultiplexer* [0..1]) $\wedge$
Notified (*thread-pool* [$*$], *reactor* [0..1]) $\wedge$
Dispatched (*reactor* [0..1], *concrete-event-handler* [$*$]) $\wedge$
Activated (*handle* [0..1], *concrete-event-handler* [0..1]) $\wedge$
Leader (*thread-pool* [0..1]) $\wedge$
First-Follower (*thread-pool* [0..1]) $\wedge$
Follower (*thread-pool* [$*$])
Register (*r,ceh,h*): $\neg$ *Registered* (*r,ceh*) $\wedge\neg$ *Owned* (*handle, ceh*) $\rightarrow$ *Registered'* (*r,ceh*) $\wedge$ *Owned'* (*h,ceh*) $\vee$
Remove (*r,ceh,h*): *Registered* (*r,ceh*) $\wedge$ *Owned* (*h, ceh*) $\wedge$ *Dispatched* (*r,ceh*) $\rightarrow \neg$ *Registered'* (*r,ceh*) $\wedge\neg$ *Owned'* (*h,ceh*) $\vee$
Promote (*r, tp*1,*tp*2,*ceh,h,sed*): *add* (*r, subst* ({*hs/sed, tp*1*/r in* $\neg$ *Dispatched* (*tp*1,*ceh*)}, *Promote* (*tp*1,*tp*2,*ceh,h,hs*)) $\wedge$ *Notified* (*tp*1,*r*)) $\vee$
Dispatch (*r,sed,ceh,h*): *add* (*r, subst* ({*hs/sed, tp/r*}, *Dispatch* (*hs,tp,ceh,h*)) $\wedge$ *Notified* (*tp,r*)) $\vee$
*Join*1 (*r,tp*1,*h,ceh*): *add* (*r, Join*1 (*tp1,h,ceh*) $\wedge$ *Notified* (*tp*1,*r*)) $\vee$
*Join*2 (*r,tp*1,*tp*2,*h,ceh*): *add* (*r, Join*2 (*tp*1,*tp*2,*h,ceh*) $\wedge$ *Notified* (*tp*2,*r*)) $\vee$
*Join*3 (*r,tp*1,*tp*2,*tp*3,*h,ceh*): *add* (*r, Join*3 (*tp*1,*tp*2,*tp*3,*h,ceh*) $\wedge$ *Notified* (*tp*3,*r*))

relations *Registered*, *Owned*, *Ready* and *Dispatched* of the combined pattern are taken from the *Reactor* pattern while the temporal relations *Activated*, *Leader*, *First-Follower* and *Follower* are taken from the *Leader/Followers* pattern. A new temporal relation *Notified* was added to the combined pattern to reflect the fact that now the leader thread notifies the *reactor* when a *handle* becomes ready but it is the *reactor*'s job to dispatch the event to the appropriate *concrete-event-handler*. Similarly, the actions *Register* and *Remove* of the combined pattern are taken from the *Reactor* pattern while the actions *Promote* and *Dispatch* are taken from the *Leader/Followers* pattern with some additions and substitutions. The actions *Join*1, *Join*2 and *Join*3 of the combined pattern are taken from the *Leader/Followers* pattern with some additions.

5. Related Work

There are many formal specification languages proposed by academia and industry. However, not all of them are suitable for specifying design patterns, mainly because design patterns are abstractions and do not need specifications catering to low-level details. Also patterns cut across module boundaries and not many specifications approaches can support such cross-cuttings. The most promising alternatives for specifying design patterns are LanguagE for Patterns' Uniform Specification (LePUS) (Eden and Hirshfeld, 1999), Distributed Co-operation (DisCo) (Back and Kurki-Suonio, 1988), Constraint diagrams (Lauder and Kent, 1998) and Contracts (Helm *et al.*, 1990). LePUS and DisCo have achieved the best ranking after an extensive comparison based on well-defined evaluation criteria (Taibi and Ngo, 2003).

As such, the BPSL is based to a certain extent on both mathematical backgrounds of LePUS and DisCo (FOL and TLA). LePUS is derived from Higher-Order Logic (HOL) and focuses only on specifying the structural aspects of design patterns. We preferred to use a small fraction of FOL because approachability was paramount in the design of the 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 was derived from TLA and was designed to specify reactive systems, which are in constant interaction with their environment and therefore have a predominant behavioral aspect. DisCo has little (almost none) support for specifying the structural aspect. The subset of TLA used in the 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 2.2.

A lot of work has been done on formally specifying software components and their combination: however, comparatively little has been done with reference to formal specification of the design pattern combination. Patterns and software components share a lot of things because they can both be building blocks of software architecture. However, the fact that patterns represent successful solutions to well-known design problems within certain contexts makes them special components. Each pattern represents a well-tested abstraction that has an infinite number of instances (implementations). For these reasons patterns are building blocks from which more reusable and changeable software designs can be built. In the following, we will only highlight the work on formally specifying pattern combination that is found to be very related to what is presented in this paper. Composition has been studied by Abadi and Lamport (Abadi and Lamport, 1993). Their results are applicable to any domain, whereas ours are specialized in pattern combination.

Saeki (2000) used the Language of Temporal Ordering Specification (LOTOS) (ISO 8807; 1989) to specify pattern combinations. The formal semantic of the LOTOS is based on the Calculus of Communicating Systems (CCS) for behavior specification and on the Algebra of Abstract Data Type (ADT) for data specification. The LOTOS was originally devised by the International Organization for Standardization (ISO) to specify the layers and their interaction for the Open System Interconnection (OSI) model. The LOTOS has been adapted in (Saeki, 2000) to be used for specifying patterns that appeared in (Gamma *et al.*, 1995) and their combination. While the LOTOS is best suited for network layers specification, its adaptation to patterns did not yield simple and clear specifications as expected by any formal specification language.

Dong *et al.* (2000) used FOL *theories* to specify the structural aspect of patterns and TLA to specify their behavioral aspect. The same techniques were used to specify pattern combinations. Each pattern is specified by an FOL theory that is derived from its signature (name, domain and range). From a structural aspect, pattern combination is performed using *name mapping*, which associates the classes and objects declared in a pattern with the classes and objects declared in the pattern combination. The specification of the behavioral aspect is done purely using TLA not a subset thereof. From a behavioral aspect, pattern combination is simply the conjunction of the TLA formula representing each of the participating patterns. Although name mapping is similar to FOL substitutions, the BPSL approach to the structural aspect specification is more rigorous as it uses a carefully chosen subset of FOL not FOL theories. This subset was chosen to cater for the properties of patterns in particular, not components in general. Moreover, the concept of elimination was added to FOL to cater for cases where some variable symbols and predicates need to be removed after combination. Furthermore, we believe TLA is too generic to be applied directly to specify the behavioral aspect of patterns but rather a subset thereof, which is specifically devised to tackle all issues and particularities of patterns. In the BPSL pattern combination (from a behavioral aspect) is not simply the conjunction of behavioral specifications of the underlying patterns, but a rigorous process that involves identifying the behaviorally dominant pattern (if any), substitutions, partial substitution and addition in temporal relation and actions as well as introducing new actions (if required).

Mikkonen (1998) used a variation of DisCo to specify the behavioral aspect of patterns. The specification of pattern combination was done using as an example the *Observer-Mediator* pattern combination. The correctness of the specification of the pattern combination is ensured using the concept of *refinement*. Actions of the pattern combination *must* refine actions from *both* patterns, which guarantee to satisfy their characteristic properties. Although the concept of *refinement* is similar to the concept of *substitution*, substitutions in the BPSL do not force the usage of actions from both patterns. Moreover, the BPSL possesses more operators than substitutions a defined in Section 3.1 such as partial substitution, elimination, addition and addition of action preconditions.

6. Conclusion

The appealing increase in network and processing speeds has increased the popularity of distributed application development to take full advantage of their inherent benefits. However, as in any other software development effort, distributed application development is facing enormous complexities and challenges. Design patterns seem to alleviate the problem by providing sound and robust solutions that have been successfully applied in previous development efforts. Current textual descriptions of design patterns and especially their combination usually lead to ambiguity and misunderstanding due to the inherent liabilities of the natural language. As patterns represent abstractions, any formal language meant to specify them should strive to achieve simplicity, accuracy and completeness. Since patterns have two complementary aspects (structural and behavioral), the BPSL was devised to combine the specification of the two aspects in order to achieve completeness. The 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 patterns accurately. The BPSL was successfully used to formally specify all cases of pattern combination. In this paper the formal specification of the *Reactor*

and *Leader/Followers* patterns as well as their combination was taken as a case study. The specification of the combination for both parts (structural and behavioral) is guaranteed to be correct because it is built on top of the existing well-tested specifications of the patterns involved in the combination. As such, instances of the pattern combination can be used in particular applications without further proof, which saves considerable efforts of fixing errors downstream in the software development process.

References

Abadi M. and Lamport L. (1993): *Composing specifications*. — ACM Trans. Programm. Lang. Syst., Vol. 15, No. 1, pp. 73–132.

Back R.J.R. and Kurki-Suonio R. (1988): *Distributed cooperation with action systems*. — ACM Trans. Programm. Lang. Syst., Vol. 10, pp. 513–554.

Cheesman J. and Daniels J. (2000): *UML Components: A Simple Process for Specifying Component-Based Software*. — Reading, MA: Addison-Wesley.

Chinnasamy S., Raje R.R. and Liu Z. (1999): *Specification of design patterns: An Analysis*. — Proc. 7th Int. Conf. *Advanced Computing and Communications, ADCOM'99*, Roorkee, India, pp. 300–400.

Dong J., Alencar P.S.C. and Cowan D.D. (2000): *Ensuring structure and behavior correctness in design composition*. — Proc. 7th IEEE Int. Conf. *Workshop on the Engineering of Computer Based Systems*, Edinburgh, Scotland, pp. 279–287.

D'Souza D.F and Wills A.L (1998): *Objects, Components, and Frameworks With UML: The Catalysis Approach*. — Reading, MA: Addison-Wesley.

Eden A.H. and Hirshfeld Y. (1999): *LePUS-Symbolic Logic Modeling of Object Oriented Architectures: A Case Study*. — Proc. 2nd Nordic Workshop *Software Architecture, NOSA'99*, Ronneby, Sweden.

Eden A.H. and Hirshfeld Y. (2001): *Principles in formal specification of object-oriented architectures*. — Proc. *IBM Center for Advanced Studies Conference, CASCON'2001*, Toronto, Canada.

Gamma E., Helm R., Johnson R. and Vlissides J. (1995): *Design Patterns: Elements of Reusable Object-Oriented Systems*. — Reading, MA: Addison-Wesley.

Helm, R., Holland, I.M. and Gangopadhyay D. (1990): *Contracts: Specifying behavioral compositions in object-oriented systems*. — Proc. ECOOP/OOPSLA'90, Ottawa, Canada, pp. 169–180

ISO 8807 (1989): *Information Processing Systems, Open Systems Interconnection-LOTOS-*. — A Formal Description Technique based on the Temporal Ordering of Observational Behavior.

Lamport L. (1994): *The temporal logic of actions*. — ACM Trans. Programm. Lang. Syst., Vol. 16, No. 3, pp. 872–923.

Lauder A. and Kent S.(1998): *Precise visual specification of design patterns*. — Proc. Europ. Conf. *Object-Oriented Programming, ECOOP98*, Brussela, Belgium, pp. 114–134.

Mikkonen T. (1998): *Formalizing design patterns*. — Proc. Int. Conf. *Software Engineering, ICSE'98*, Kyoto, Japan, pp.115–124.

Rambaugh J., Jacobson I. and Booch G. (1998): *The Unified Modeling Language Reference Manual*. — Reading, MA: Addison-Wesley.

Saeki M. (2000): *Behavioral Specification of GOF Design Patterns with LOTOS*. — Proc. *IEEE Asia Pacific Software Engineering Conference, APSEC'2000*, Singapore, pp. 408–415.

Schmidt D.C., Stal M., Rohnert H. and Buschmann F. (2000): *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. — New York: Wiley.

Taibi T. and Ngo D.C.L (2001): *Why and how should patterns be formalized*. — J. Object-Oriented Programm., Vol. 14, No. 4, pp. 8–9, 101 communications.

Taibi T. and Ngo D.C.L (2002): *Formal specification of design patterns—A comparison*. — Tech. Rep., Faculty of Information Technology, Multimedia University.

Taibi T. and Ngo D.C.L (2003): *Formal specification of design pattern—A balanced approach*. — J. Object Technol., Vol. 2, No. 2, pp. 127–140.

Vlissides J.M. (1997a): *Multicast*. — C++ Report, Sep. 97, SIGS Publications.

Vlissides J.M. (1997b): *Multicast - Observer = Typed Message*. — C++ Report, Nov.-Dec. 97, SIGS Publications.

Received: 4 July 2002
Revised: 6 January 2003
Re-revised: 31 March 2003