Design Verification Patterns

John Knudsen, Anders P. Ravn, and Arne Skou

Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg, Denmark
apr@cs.aau.dk

Abstract. Design Verification Patterns are formal specifications that define the semantics of design patterns. For each design pattern, the corresponding verification pattern give a set of proof obligations. They must be discharged for a correct implementation of the pattern. Additionally there is a set of properties that may be used in the design and verification of applications that employ the pattern. The concept is illustrated by examples from general software engineering and more specialised properties for embedded software.

1 Introduction

Engineers design; thus software engineering is a discipline that systematizes knowledge about procedures for designing software. This is evident from the structure of Dines Bjørner's volume on the subject [2]: After a careful analysis of the application domain and a systematic elicitation of requirements, the remaining task is design with implementation. Like in any other engineering discipline software design is based on reuse of patterns and well known components. However, unlike other disciplines, software engineering does not systematically use the patterns and components to analyse properties of the resulting system. Software is generally built without systematic analyses. Throughout the ProCoS project [8,14] it was the ambition to improve on this state of affairs, and through numerous case studies, we demonstrated that is was feasible, see for instance [24,23].

Yet, application of formal techniques have not spread dramatically, and it is rather clear that it is so difficult that it will remain a specialist activity even with better integrated notations like those developed in Oldenburg [21,13] and at UNU/IIST [9,16]. Perhaps a remark from control engineering colleagues helps to clarify, how difficult it is to work rigorously from basics: "Either you choose a PID controller or you have to get a PhD-controller." Yet, we cannot expect every engineer to have skills at a PhD level, therefore we must rely on standard components that we know well, and where there are standard procedures for tuning them and *checking their properties*. Design Verification Patterns is an attempt at defining properties for the standard design patterns for software engineering.

C.B. Jones, Z. Liu, J. Woodcock (Eds.): Bjørner/Zhou Festschrift, LNCS 4700, pp. 399–413, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The idea is rather natural, and one may wonder why it has not been done already. Here it helps to look at history: In the 1990ies, software was coded using languages like `c`, and systematic designs were very hard to discover in the programs that resulted from this activity. Components were statements, and analysis would be at a similar low level; it would focus on programming language semantics and the corresponding program correctness theories as for instance consolidated in Hoare and He's "Unified Theory of Programming" [11].

Since then, the demand for more software to increasingly efficient computers that find applications in the most diverse areas - ubiquitous computing - means that the level of abstraction is lifted. There is increased use of object oriented languages and design notations like UML [25,7], beginning experiments with reuse of components [28], and concern for architecture [1]. These ideas are combined with formal techniques and gains increasing popularity through efforts of Meyer [19], which are continued in tool developments like JML [3].

Thus, modern software relies on datatypes and common functions as embodied in the standard class libraries of objected oriented programming languages. A corresponding level of structuring constructs comes with the practical use of design patterns [15,6]. They may be the "PID"s for the practicing software engineer, but what are their properties, and how may these properties be used to analyze applications?

A Design Verification Pattern. As an introduction to properties of design patterns, we may look at the ancestor to all design patterns: The *procedure pattern.* A (side-effect free) procedure p has an input or value parameter x and computes an output or result y:

$$p(value\ x;\ result\ y)$$

With axiomatic semantics, we know that it may be fully specified in terms of two predicates, a pre- and a post-condition: $p.pre(x)$ and $p.post(x, y)$, where the pre-condition specifies the domain of the procedure, and the post-condition specifies the effect in the form of an input-output relation.

Analysing Properties. It is also clear that an implementer of p has an obligation to *guarantee* the post-condition, but only when one can *rely* on the pre-condition. This is the basis for verifying correctness of the component p. From our point of view, it is also the *design verification pattern.* The lemmas that can be used in an application. Just before a call $p(e, r)$, the application must *assert* that the the procedure can relay on the pre-condition with e for x, $p.pre[e/x]$. Then, after the call, it is legitimate to *assume* the post-condition with a similar substitution,$p.post[e/x, r/y]$, and use it in an analysis of application properties.

Much research in verification has focused on the obligations of the implementors. We are less concerned with this, because with increasing reuse, components are developed by specialists - the PhDs, who should be able to deal with the task. In contrast, the components are used in many contexts, so the lemmas formulated in the assert and assume conditions are a higher level start for verifying

applications. They may form the basis for harnessing theories for tool support [17], a point we will return to in the concluding Section 4.related work.

Beyond Functional Aspects. In the more complex setting of objects or components, the verification must go beyond pure functional properties as expressed in the pre- post-condition paradigm. There is a state component as well that enters in the post-condition and which satisfies some *invariant*. Furthermore, many applications are reactive systems where the properties are protocols, that is constraints on *behaviours* of concurrent processes. An additional aspect occurs with embedded systems where *real-time properties* are important. In Section 2 we will introduce suitable notation for expressing such non-functional aspects, before we exemplify in Section 3 with conventional design patterns and extends it with a discussion of timing properties which are important for embedded software systems.

2 Background

In the following we introduce verification patterns more formally after defining the notations that are used to define properties. Settling on notation is a matter of preference, but it is important that the chosen notation conveniently can express what is required and that it is well established so that it is easily understood. The most widely used notation for design patterns today is UML [25] so UML class diagrams are the syntax in the following. To enable formal specification and reasoning, the UML diagrams must be given semantics. The formalism chosen to serve this purpose is the CSP-OZ-DC combination of Oldenburg [12,21,20] as elaborated in Hoenicke's dissertation [13]. The CSP [10] part hereof allows specification of processes, Object Z (OZ) [26] allows specification of functional aspects for operations on objects, and Duration Calculus (DC) [31,30] enables reasoning about time aspects.

OZ. Here, we are not going to go into syntactical and semantical details of OZ, but just note that a class C corresponds to a Z schema, a method m to an operation on the schema, and that an object o of the class C is a *reference* to a value of the schema.

CSP. The *communication events* are elaborated such that a channel is associated with each public method and thereby with the corresponding schema operation. A method call to an object, say with input parameter x, actual argument e, and output parameter y of type T, is written $o.m!(x == e)?(y : T)$, cf. [13]. Otherwise, we have the usual syntax for CSP processes which syntactically are added as constraints to OZ schemas, or in UML as constraints in a *responsibility* part of a class or object. For convenience, we list the CSP operators:

$$P \ ::= \ \text{STOP} \mid \text{SKIP} \mid ce \rightarrow P \mid P \Box P \mid P \sqcup P \mid P \| P \mid P; \ P$$

where STOP is the deadlocked process, SKIP is the terminating process, $ce \rightarrow P$ communicates event ce and continues as P, $P \Box P$ is external choice, $P \sqcup P$ is non-deterministic choice, $P \| P$ is parallel composition, and $P;\ P$ is sequential composition. As usual, recursive definition of processes is allowed.

Duration Calculus. Duration Calculus formalizes dynamic systems properties. The basis is the well-known *time-domain model*, where a system is described by a collection of *states* which are functions of time (the non-negative real numbers). The state names are here the variables in a given schema, which clearly vary over time.

A *behaviour* of a system is thus an assignment of state functions to the names of elementary states, An *observation of a behavior* is a restriction of such an assignment to a bounded interval; it can be illustrated by a timing diagram. Boolean values and thus the value of state predicates are by convention represented by 0 (*false*) and 1 (*true*).

For a given observation interval $[b, e]$ of a predicate P, the *duration*, denoted $\int P$ is simply the integral $\int_b^e P(t)\,dt$; it measures the fraction of time P holds in the interval.

Duration terms are built from durations, logical variables and real numbers and closed under arithmetic operators and arithmetic relations. *Duration formulas* D are built from duration terms of Boolean type and closed under propositional connectives, the Interval Temporal Logic [22] "chop" connective, and quantification over rigid variables and variables of duration terms.

A duration formula D *holds* in $[b, e]$ if it evaluates to true. For the predicate P, it is obvious that it holds (almost everywhere) in the interval, just when the duration $\int P$ is equal to the length of the interval. The length is the duration of the constant function 1 ($\int 1$). This duration is often used, so it is abbreviated (ℓ), pronounced 'the length'. The property that P holds is thus given by the atomic formula $\int P = \ell$. This holds trivially for a point interval, so we consider proper intervals of a positive length. These two properties are combined in the abbreviation

$$\lceil P \rceil \ == (\textstyle\int P = \ell) \wedge\ (\ell > 0)$$

read as 'P holds'.

Given formulas D and E, the binary "chop" connective can combine them to $D\,;E$ which holds on $[b, e]$ when there exists a m such that D holds on $[b, m]$ and E holds on $[m, e]$. A simple example is the valid equivalence $\lceil P \rceil = \lceil P \rceil\,;\lceil P \rceil$.

With CSP, we include event based reasoning by defining that for an event ce, the formula $\uparrow ce$ holds exactly when the event occurs at the beginning of the interval. Thus we can specify an interval where ce does not occur in the open interval by the counterexample formula

$$\overline{ce} == \neg(\ell > 0\,; (\uparrow ce)\,; \ell > 0))$$

2.1 Verification of Design Patters

In describing the relation between the syntactic language of UML and the semantics of the formal description language package from Oldenburg, the CSP-OZ-DC it is now possible to illustrate the idea of using verification patterns in the software development process graphically, as seen in Figure 1.

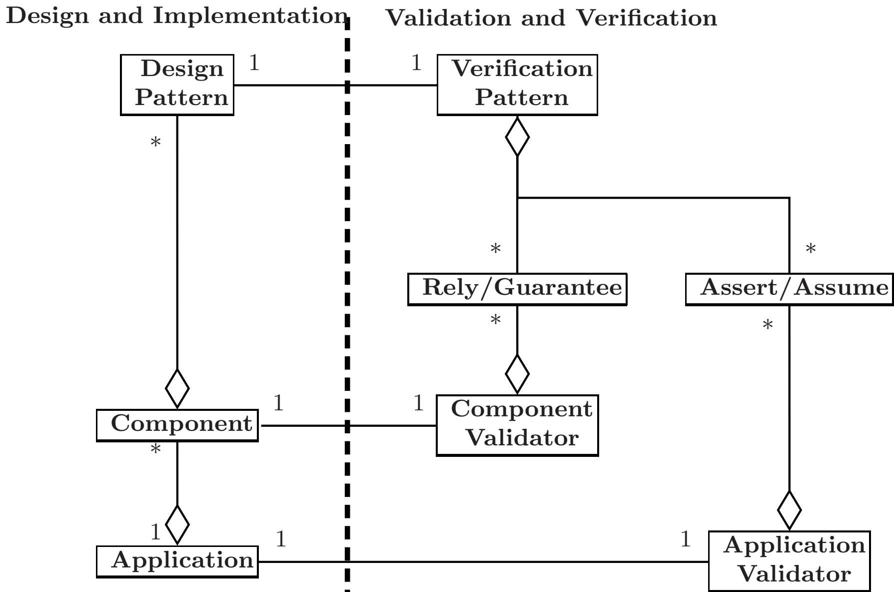

Fig. 1. The framework of Component Based Development using Design Patterns and Verification Patters

Figure 1 represents the framework for Component Based Development using Design Patterns and Verification Patterns. The framework extend what is considered ordinary software development efforts by explicitly emphasising the use of design patterns and by illustrating the associated validation possibilities that verification patterns offer.

Design patterns are usually applied in a context similar to the one illustrated in the *Design and Implementation* part of Figure 1. Design patterns provides guides on how to develop good designs to well known design challenges and components that must address one or more of these challenges can then be designed using the respective design patterns. That the component is designed and implemented using certain design patterns can be useful for the system developer that later will embed the component in a system design. Most design patterns, however, contains more useful information that can be exploited, as illustrated in the *Validation and Verification* part of Figure 1. For each design pattern a verification pattern, which is a range of rely/guarantee pairs, can be specified reflecting the different aspects of the pattern. The specification can

be used either as test and verification of the component, or in a test driven applicaition development process model as assert/assume pairs for verification, monitoring or for developing tests.

2.2 Related Work

There is a rich literature on verification, and patterns are implicit in many proof rules, yet the idea of combining architectural patterns with verification occurs, to our knowledge for the first time in [4,5]. It develops application specific pattern for collision avoidance. A somewhat similar idea is to develop refinement rules or conditions for design patterns; this is explored in [18]. Finally we mention [27] which approaches design patterns with the same mission to delimit the responsibilities of the developer and define the rewards for the application programmer. Their proposal for formalization, however, does not distinguish between aspects and thus require coding behavioural properties as state invariants.

3 Patterns

We begin with one of the most used and most simple design patterns; the Singleton pattern. A good example of its application is the control module for a ship's rudders. A modern ship is controlled from several stations, but it is essential that the rudders have one and only one common interface, such that clients that needs access to the rudders access the same software unit. If different clients have or create their own representation of the rudders, the representations are hard, if not impossible to keep consistent. A correct instantiation of a Singleton pattern eliminates such behaviour.

3.1 Singleton Pattern

The intent of the Singleton pattern is to ensure that there can only be one instance of the Singleton class and to provide global access to this instance [6].

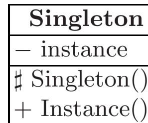

Fig. 2. The Singleton pattern as an UML class diagram

Figure 2 shows the UML class diagram for the Singleton pattern. Note that the instance attribute has a − prefix, indicating that it is private to the class, i.e. only accessible through methods/operations of the class. For the Singleton class a protected constructor operation, ♯ Singleton(), is given and the public operation + Instance().

The pattern description provide information that can be very useful in verifying that an actual design or implementation actually is in accordance with the intent of the pattern.

Implementation: A Singleton is usually implemented as a class with a *protected* constructor as the only constructor. This should prevent external use of *new* to generate more instances. The instance of the Singleton is created first time the Instance() method is called. This call looks up the private instance attribute to see if it exists, and if it is not an instance is created, assigned and returned to the caller. For future calls the instance is returned.

Functional Verification Conditions: The Singleton pattern is a creational pattern, where there as such is not much to verify. There is an *invariant* stating that the instance reference is valid and that it is not changed unless it is *null* by the Instance operation. The pre-condition for the Instance operation is trivially *true* and the post-condition states that the result of the operations is a copy of the reference to this instance.

Behavioural Verification Conditions: There is one specified behaviour

$$main == s.Instance!(y == s.instance) \rightarrow main$$

When applying a singleton pattern with class s, we do not have to take any special precautions, the *assert* is trivially *true*. At any point, we can *assume* that there is at most one instance of the class and that this instance remains the same, i.e. it is not overwritten. Thus combining the functional and behavioural condition, we can prove that an application satisfies the property that

$$(s.Instance?y : Singleton \rightarrow P(y))\|((s.Instance?y : Singleton \rightarrow Q(y))$$

is equivalent with

$$(s.Instance?y : Singleton \rightarrow (P(y))\|Q(y))$$

Thus, we can eliminate or introduce (using CSP laws) multiple calls of *Instance*.

The idea of exploiting a design using design patterns to extract verification lemmas is in Example 1 made less abstract as the use of a Singleton patterns is embedded in a design.

Example 1. On a modern ship the rudder can be manipulated and monitored from various stations on the ship. Here it is relevant to consider the rudder controller as a Singleton. In this example the rudder can be manipulated from the ship bridge and from the machine room.

In Figure 3 the UML diagram illustrates a design where the classes **Bridge** and **Machine** both have a **RudderSingleton** object. Although UML allows specification of the arity between objects by annotating the edges between them, in Figure 3 by 1's at the RudderSingleton class, this only states that one Bridge

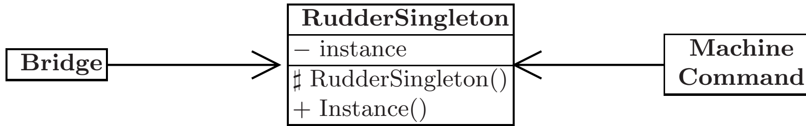

Fig. 3. UML class diagram showing a rudder control design

object has one RudderSingleton and one Machine object should have one RudderSingleton and not that it should be the same object. However, the verification pattern allows us to deduce that.

Also, depending on the structure of the application, it may show us that the Machine and the Bridge has simultaneous access to the rudders, and that may lead to some conflicts about who controls the ship; but such a source conflict is not handled by the Singleton pattern. It only ensures that there is a single control object with atomic operations.

Timing Verification Condition: In embedded applications, the only timing condition we can think of in connection with the Singleton pattern is the (worst case) execution time for the method. It produces a rely condition, which is an invariant that the application has to satisfy:

$$(\uparrow \mathit{Instance} \wedge \ell > 0)\,;(\uparrow \mathit{Instance}) \Rightarrow \int \mathit{Active}(p) \geq \mathit{WCET}_{\mathit{Instance}}$$

here, *Active* is for each process a state variable which is true exactly when the process is executing. A naive formulation would use ℓ for $\int \mathit{Active}(p)$ but that would assume a dedicated processor for each process. The *Active* variable is manipulated by scheduling mechanisms which may be specified by duration formulas [29].

3.2 Observer Pattern

A more intricate example is the Observer pattern. It is often used in embedded systems to signal changes in an environment.

Intent: The intent of the observer pattern is to define a one to many dependency on objects, so that when one object changes state, all its dependents are notified.

Motivation: Partitioning leads to a need to maintain consistency between objects without tight coupling for reusability. Observers are updated when the subject changes state. The key objects in the Observer pattern are *observer* and *subject.* A subject may have any number of dependant observers, where all observers are notified when the subject has changed state. The subject sends its notifications without knowing its observers.

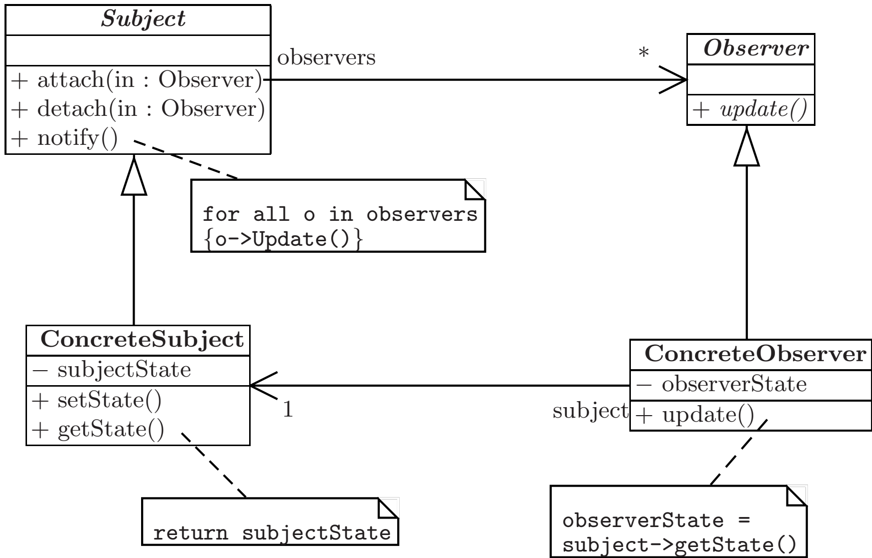

Fig. 4. A UML class diagram of the Observer pattern

Structure: The class structure of the Observer pattern is illustrated by the class diagram in Figure 4. The Observer pattern is modeled with the above mentioned classes *Subject* and *Observer* as abstract classes with a directed association from *Subject* to *Observer*, indicating that the subject is capable of calling the *Update* operation on observer objects.The two abstract classes each have a concrete counterpart, *ConcreteSubject* and *ConcreteObserver*, that inherit from the respective abstract classes. Between the *ConcreteSubject* and *ConcreteObserver* classes a directed association towards the *ConcreteSubject* lets concrete observer objects call the *getState* operation on concrete subjects to get its state.

Participants: Subject knows its observers. Any number of observers can observe a subject. *Observer* defines an updating interface for objects that should be notified of changes in a subject. *ConcreteSubject* stores the state of interest to the *ConcreteObserver* and sends a notification to its observers when the state is changed. The *ConcreteObserver* maintains a reference to a *ConcreteSubject* object and stores the state of it to stay consistent. A sequence diagram illustrating interactions of the participants is given in Figure 5.

Verification Conditions for the Observer pattern: The structure of the Observer pattern, as given in Figure 4, imply a CSP-OZ translation as in Figure 6 which gives the functional verification conditions. The *Observer* and *Subject* classes are both specified as abstract classes with an association that is navigable form *Subject* to *Observer*. The association implies a data structure in a *Subject* object containing *Observer* objects. This is to register the objects that attach

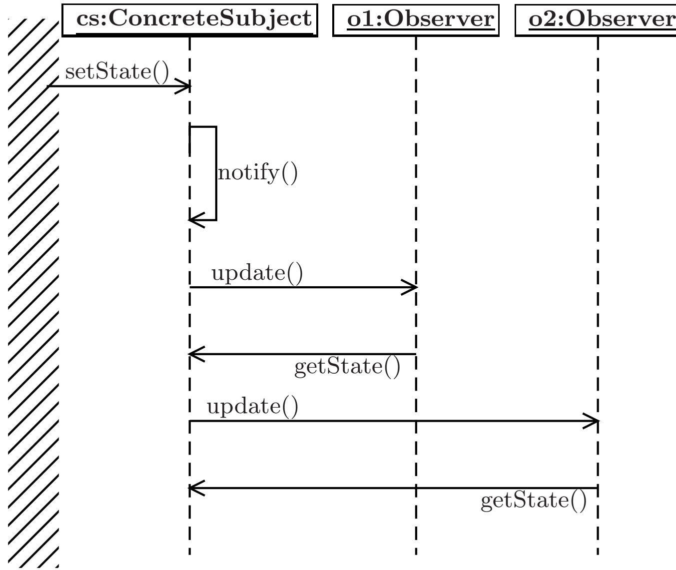

Fig. 5. Interactions between the participants in the Observer pattern

themselves as observers. As the classes are declared abstract this is not really going to be the case, but the concrete "children" objects, objects of the classes that inherit from the respective abstract classes have to posses these properties.

The classes *ConcreteSubject* and *ConcreteObserver* are the classes that inherit from respectively the *Subject* and *Observer* classes. The *ConcreteObserver* is related to the *ConcreteSubject* by an association that is navigable towards the *ConcreteSubject*. This means that a *ConcreteObserver* object will have an attribute of type *ConcreteSubject*, which store the object to which it is attached.

In [6] the *notify* operation has a note which says that the operation should implement a sequential call of the *update* operation on all attached observers. The *update* operation in the *Observer* class has no attached specification, and do as such only specify an interface to the concrete observer. Perhaps a use of the UML interface class to represent the observer would have been more appropriate, as the *update* operation is declared as abstract, leaving it to the inheriting classes to implement the operation body.

Behavioural Verification Condition The original specification of the Observer pattern do not provide any information on the events of a process encapsulating the pattern. The behaviour that is *guaranteed* is

$$\texttt{main} \stackrel{c}{=} \mathit{Subject.setState} \rightarrow \mathit{Subject.Notify} \rightarrow (\|_x : o \bullet x.\mathit{Update} \rightarrow \mathit{SKIP});\ \texttt{main}$$

The purpose of the observer pattern is as stated to notify observers of state changes to some subject. This assumption can be analyzed by investigating

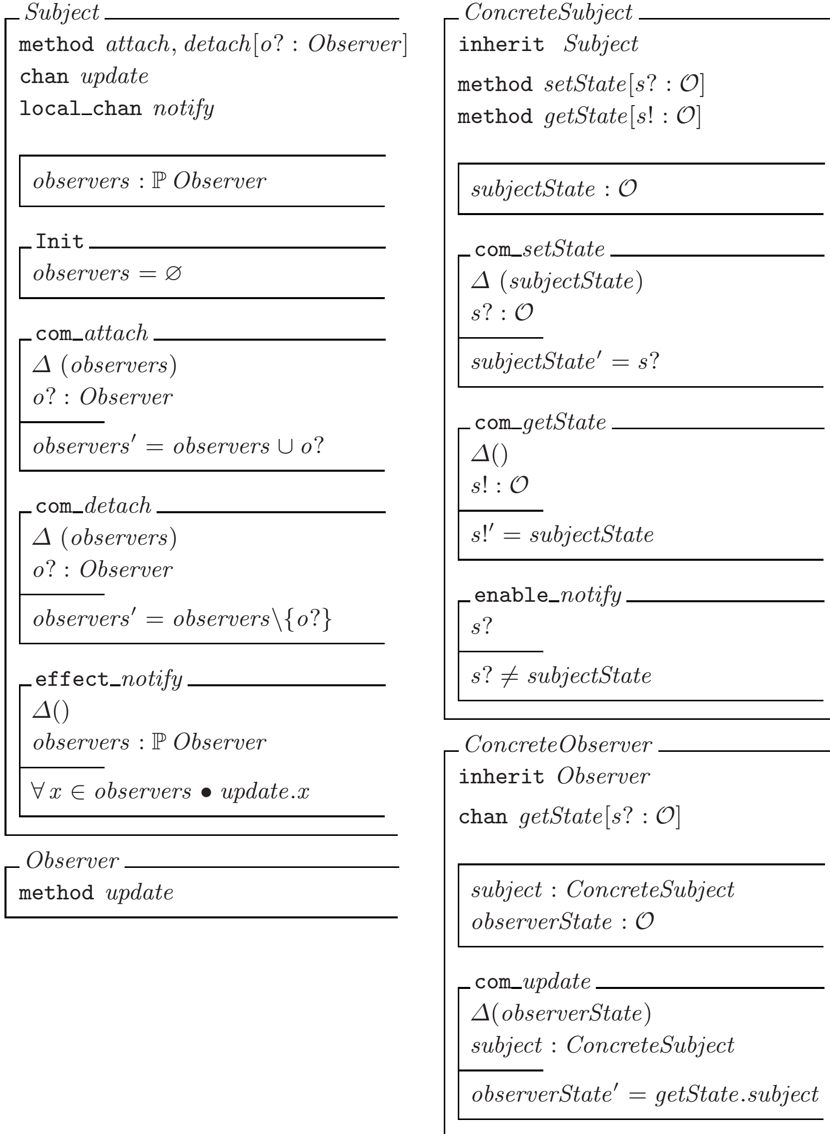

Fig. 6. The structure of the observer pattern as CSP-OZ schemata

possible method call sequences. For observers $o_1, \ldots, o_n$ and a concrete subject object cs a state change must be recognized as following sequence:

$$cs.Attach(o_1);\ cs.Attach(o_3);\ \ldots;\\ cs.setState();\ cs.Notify();\ o_1.Update();\ o_2.Update();\ \ldots.$$

Popular descriptions of the pattern indirectly let readers assume that the consistency is guaranteed by the proposed design and the design pattern is motivated by a need to keep observer states consistent with subject states. Yet, the design pattern description puts no constraints on observers' *getState*() calls on the subject. This may lead to inconsistency. If o_s is a slow observer attached to *cs*, a concrete subject and o_s as a data logger must register all states of the subject, the following possible sequence shows that o_s is not dependable:

$$\ldots;\ cs.setState();\ cs.Notify();\ o_s.Update();\ \ldots;\ cs.setState();\ cs.getState()$$

As observed in the above sequence, it is possible that the state of the subject changes before the observers have finished updating. The description and structure of the observer pattern as stated above has data pull characteristics, where for i.e. embedded systems and control systems data push characteristics might be more relevant.

The problem of data push versus data pull of the observer pattern was already addressed in [6], where it is noted that modifying the *Update*() method to take the subject state as parameter actually would convert the pull characteristics to push characteristics. Such a change to the pattern allow a stronger assumption on the relation between subject state and observer state. We have the sequence after interested objects have been attached:

$$\ldots;\ cs.setState();\ cs.Notify();\ o_1.Update(subjectState);\ o_2.Update\ (subjectState);\ \ldots$$

and

$$cs.subjectState = o_1.observerState = \ldots = o_n.observerState$$

Timing Verification Conditions The observer pattern corresponds to an aperiodic task in a hard-real time setting, and it is well known that no guarantees can be given for handling all events of such a task. One has to assume that it is sporadic, that is the application has to guarantee a minimum interarrival time T, as formulated in the invariant:

$$(\uparrow setState \land \ell > 0)\,;(\uparrow setState) \Rightarrow \ell \geq T$$

Furthermore, the worst case execution time for Update leads to a further assumption on the application analogous with the one for the Singleton Pattern, and finally, the number of attached tasks must be used to define an assumption linking T and the worst case execution time for the set of Updates.

If the ConcreteObserver is designed according to the data pull variant of the Observer Pattern, following constraint must hold to ensure consistency between a concrete subject and concrete observers:

$$(\uparrow Update \land \ell = 0)\,;(\overline{Update} \land \overline{getState})\,;(\uparrow getState \land \ell = 0) \Rightarrow \ell \leq T$$

It says that the elapsed time between two consecutive *Update*, *getState* events is less than mimnimum interarrival time.

3.3 Summary

We have illustrated that it is feasible to specify both functional, behavioural and timing conditions that form design verification patterns. A few other patterns have been investigated and give similar reasonably succinct conditions. Yet, there is much more work to do. Some particularly interesting questions are to what extent the verification patterns can be made complete for a given design pattern, that is they give a precise characterization of the pattern. For the functional and behavioural properties, we think they can be made complete, however, we are not sure that that can be done for the timing properties, because these are dependent on the underlying execution platforms.

4 Conclusion

We have developed the concept of a verification pattern, the semantic twin to syntactic design patterns. They are based on conventional rely-guarantee conditions for an implementation which are used as assert-assume lemmas for the application that uses them. The concept has been illustrated with functional, behavioural and timing specifications for conventional patterns.

Discussion. The application of Design Patterns in practical design of embedded systems is still at a early stage but as the complexity increase and the object oriented paradigm gains ground in this field, so will the application of technologies known from this field. A substantial part of the research in Design Patterns has focus on efficient solutions to architectural design challenges, and although designers of embedded systems are faced with such challenges also, more effort on researching Design Patterns for the special challenges that characterise embedded systems is needed.

In a case study of control software for marine diesel engines, that spurred our research of Verification Design Patterns, we encountered behavioral challenges typically encountered in control engineering that could be candidates for Design Patterns and Verification Design Patterns in embedded software systems.

One such challenge was the modelling of the *JetAssist*, a system that assist a turbo charger in optimizing the the fuel consumption and reduce CO_2 emission levels. For the JetAssist the specification explicitly stated that the system should have a *hysteresis* behaviour, i.e. the behavior of the system must by increasing and decreasing temperature, in respective overlapping intervals displaying different temperatures. That is, by measuring the temperature alone, is is not possible to predict the behavior of the JetAssist. Hysteresis is known from a variety of control and processing systems. Around this hysteresis pattern we found a number of well-known conventional patterns, exemplified by those described in this paper. It was clear to us that if we are to prove proerties of the *JetAssist* in a way that is comprehensible to engineers, we would have to structure the arguments - thus the patterns.

Acknowledgement. The first author wishes to thank CISS for funding his work on the topic, and menbers of the "Correct System Design" group, in particular Professor Olderog, for hospitality during extended stays. The second and third author acknowledge the inspiration fromk CISS' industrial collaborators to pursue this line of research.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (1999)
2. Bjørner, D.: Software Engineering. In: Bjørner, D. (ed.) Domains, Requirements and Software Design. Texts in Theoretical Computer Science, vol. 3, Springer, Heidelberg (2006)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
4. Damm, W., Hungar, H., Olderog, E.-R.: On the verification of cooperating traffic agents. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 77–110. Springer, Heidelberg (2004)
5. Damm, W., Hungar, H., Olderog, E.-R.: Verification of cooperating travel agents. International Journal of Control 79(5), 395–421 (2006)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Reading (1995)
7. Object Management Group. Unified Modeling Language: Superstructure, version 2.0, final adopted specification, `http://www.omg.org/uml/, formal/05-07-04, 2005`
8. He, J., Hoare, C.A.R., Fränzle, M., Müller-Olm, M., Olderog, E.-R., Schenke, M., Hansen, M.R., Ravn, A.P., Rischel, H.: Provably correct systems. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems. LNCS, vol. 863, pp. 288–335. Springer, Heidelberg (1994)
9. He, J., Li, X., Liu, Z.: rCOS: A refinement calculus for object systems. Theoretical Computer Science 365(1-2), 109–142 (2006)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
11. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
12. Hoenicke, J., Olderog, E.-R.: Combining Specification Techniques for Processes Data and Time. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 245–266. Springer, Heidelberg (2002)
13. Hoenicke, J.: Combination of Processes, Data, and Time. PhD thesis, Fachbereich Informatik Universitt Oldenburg (2006)
14. Langmaack, H., Ravn, A.P.: The procos project: Provably correct systems. In: Bowen, J. (ed.) Towards Verified Systems. Real-Time Safety Critical Systems, ch. Appendix B. vol. 2, Elsevier, Amsterdam (1994)
15. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice-Hall, Englewood Cliffs (2001)

16. Liu, Z., He, J., Li, X.: rCOS: A relational calculus of components. In: Mathematical Frameworks for Component Software, pp. 207–238. World Scientific, Singapore (2006)
17. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. In: Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), November 2006 (2006) (An extended version is found as UNU-IIST Technical Report 335, August 2006)
18. Long, Q., Qiu, Z., Liu, Z., Shao, L., He, J.: POST: a case study for an incremental development in rCOS. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 485–500. Springer, Heidelberg (2005)
19. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
20. Meyer, R., Faber, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 332–346. Springer, Heidelberg (2006)
21. Möller, M., Olderog, E.-R., Rasch, H., Wehrheim, H.: Linking CSP-OZ with UML and Java: A case study. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, Springer, Heidelberg (2004)
22. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. Computer 18(2), 10–19 (1985)
23. Olderog, E.-R., Ravn, A.P., Skakkebæk, J.U.: Refining system requirements to program specifications (chapter 5). In: Heitmeyer, C., Mandrioli, D. (eds.) Formal Methods in Real-Time Systems. Trends in Software-Engineering, pp. 107–134. Wiley, Chichester (1996)
24. Rischel, H., Cuellar, J., Mørk, S., Ravn, A.P., Wildgruber, I.: Development of safety-critical real-time systems. In: Bartosek, M., Staudek, J., Wiedermann, J. (eds.) SOFSEM 1995. LNCS, vol. 1012, pp. 206–235. Springer, Heidelberg (1995)
25. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modelling Language Reference Manual. Addison-Wesley, Reading (1999)
26. Smith, G.: The Object-Z specification language. Kluwer Academic Publishers, Norwell, MA, USA (2000)
27. Soundarajan, N., Hallstrom, J.O.: Responsibilities and rewards: specifying design patterns. In: Proceedings 26th International Conference on Software Engineering, ICSE 2004, May 2004, pp. 666–675. IEEE Computer Society Press, Los Alamitos (2004)
28. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1997)
29. Zhou, C., Hansen, M.R., Ravn, A.P., Rischel, H.: Duration specifications for shared processors. In: Vytopil, J. (ed.) Formal Techniques in Real-Time and Fault-Tolerant Systems. LNCS, vol. 571, pp. 21–32. Springer, Heidelberg (1991)
30. Zhou, C., Hansen, M.R.: Duration Calculus: A Formal Approach to Real-Time Systems. In: Monographs in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2004)
31. Zhou, C.C., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters 40(5), 269–276 (1991)