Formalizing Design Patterns

(pp. 115–124, Proc. ICSE'98, IEEE Computer Society Press, 1998)

Tommi Mikkonen
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553
FIN-33101 Tampere, Finland
+358 3 365 3815
tjm@cs.tut.fi

ABSTRACT

Design patterns facilitate reuse of good design practices. They are typically given by using conventional notations that lack well-defined semantics and, therefore reasoning about their behaviors requires formalization. Even when formalized, conventional communication abstractions may lead to too laborious formalizations when addressing the temporal behavior of a pattern as a whole instead of behaviors local to its components. We show that rigorous reasoning can be eased by formalizing temporal behaviors of patterns in terms of high-level abstractions of communication, and that by using property-preserving refinements, specifications can be naturally composed by using patterns as building blocks.

KEYWORDS

Formal methods, temporal behaviors, design patterns

1 INTRODUCTION

Patterns [3, 7] facilitate reuse of well-established solutions when known problems are encountered, by supporting the use of abstractions that are above the level of individual classes and instances [6]. Therefore, by using patterns, tried and approved decisions, applicable at a level of abstraction higher than that of single objects, are naturally adopted. In practice, this means that systems are based on well-defined components, offering an option to use such components as building blocks for more complex systems [6].

According to [3], *Design patterns* are a subset of patterns that provide a scheme for refining subsystems or components of software systems, or relationships between them, thus facilitating reuse of cooperation of several components. In general, design patterns describe commonly recurring structures of communicating components that solve general design problems within particular contexts, and capture solutions that have been developed and evolved over time [7]. They are typically given in terms of programming-level abstractions, or by using representations that lack rigorous definitions of temporal behaviors. Therefore, temporal properties of patterns cannot usually be reasoned about without going to the level of implementation details. Although the act of writing a formal specification is claimed to catch errors, omissions, and ambiguities early in the design process [5], we wish to provide a possibility to reason about the temporal behavior of a system even when the associated specification is incomplete, or abstract with respect to communication primitives. For pattern-oriented development, a rigorous but practical formalization, which pays special attention to behavioral modularity, is therefore needed to express temporal behavior of patterns and systems utilizing patterns as their skeletons. Steps to this direction have already been proposed, for example in [8, 1].

The rest of this paper discusses a way to formalize temporal behaviors of design patterns, paying special attention to their natural utilization when composing specifications of complex systems. The method used for the formalization allows rigorous reasoning, and also supports a software engineering view to the development. The focus is on the mechanisms facilitated by the method, that enable formalization of patterns and their usage. Although we approach the formalization in terms of examples, the underlying principles are applicable to any pattern that involves inter-object cooperation.

The way we proceed is the following. Sections 2 and 3 briefly introduce the method used for formalization, and use it to define *Observer* pattern given in [7]. Section 4 demonstrates how to create more complex specifications by combining patterns that have already been formalized, using *Managed Observer*, a derivative of *Observer* pattern, as an example. In Section 5, this combination of patterns is instantiated into a form where concrete data is used. Section 6 summarizes the properties that allow us to consider the method as rigorous but practical for formalizing design patterns and their usage.

2 THE DISCO METHOD

DisCo is a specification method [9] intended for specification and modeling of interactions at a high-level of abstraction [2]. The formal basis of the method is in Temporal Logic of Actions [14], and many of the ideas that are embedded in the method are similar to those of *UNITY* [4]. The language used for composing specifications is textual, but graphical animation is provided by the associated tool [16]. In this context, the goal is to introduce the main principles of the method, and therefore, notations reflect the underlying logic discussed in [12], rather than the actual DisCo language.

2.1 Classes, Relations, and Actions

A DisCo specification is a definition of (a pattern for) a system. In each system, the developer can introduce *classes*, *relations* and *actions*. Classes are formulas defining the form for possible *objects*. For instance, a class C can be defined as

class $C = \{x\}$,

where x is an untyped variable. With this definition, each object o of class C contains a local variable denoted as $o.x$.

Relations are used for associating objects with each other. They are defined in the format

relation $(n){\cdot}R{\cdot}(m)$: $C \times D$,

where relation R associates n instances of class C with m instances of D. Asterisk (*) is a shorthand for any possible number of instances.

Actions are atomic units of execution, which can be understood as multi-object methods. An action consists of a list of required participants and parameters, an enabling condition, and the definition of state changes caused by an execution of the action. For example, an action A can be given as

$$A(c{:}C;\ i){:}$$
$$\quad i \neq c.x$$
$$\rightarrow\ c.x' = i,$$

where c is a *role* for an object in class C, and i denotes an untyped value given as a parameter. Expression $i \neq c.x$ is the enabling condition under which the action can be executed. Unprimed and primed variables refer to the values of variables before and after the execution of the action, respectively, thus defining the state change caused by an execution of the action. The convention is used that variables implicitly preserve their values, unless otherwise explicitly denoted.

Participants, i.e., objects that take a role in an action, and parameters, i.e., plain variables that denote individual values, are nondeterministically selected from those that are suitable. For example, the above action is enabled for values of parameter i that are different from the value of $c.x$.

Actions are executed in an interleaving manner. If there are several actions that could be executed, one is nondeterministically selected from those that are enabled. This model of execution liberates the developer from defining who is responsible for initiating actions, and sets the emphasis on the cooperation of objects.

Each DisCo specification is a description of the temporal behavior of a closed system, which can be observed but not affected from outside. The behavior of the system is thus completely defined by the specification, without any implicit flows of control. In order to facilitate genericity with respect to objects, a specification does not require the developer to fix the numbers of objects that are needed. For actions, genericity is achieved by stating that objects belonging to certain classes are needed for an execution instead of identifying the actual objects that are involved.

2.2 Modularity

DisCo modularity consists of applying *superposition* [9] to existing specifications, ensuring by construction that *safety properties* ('nothing bad will ever happen') are preserved. Each refinement step can introduce a relatively small set of global aspects instead of a large number of aspects local to an object. Such a refinement can be understood as a system-wide layer that introduces slices of objects, which resemble program slices [17]. However, unlike program slices, these slices of objects aim at construction of a specification, not at decomposition of an existing system.

As the unit of modularity is a behavioral layer rather than an individual object or class, the emphasis is on the behavior of the system as a whole instead of local behaviors of independent objects, which is the case in conventional object-oriented approaches.

In practice this means that, when refining specifications, new classes and variables can be introduced, and operations affecting the new variables can be added. New operations are given as new actions, or as augmentations to existing ones, with an option to add new conjuncts to enabling guards as well.

Classes are extended by using the format

class $C = C + \{y\}$,

and action refinements are given in the format

$B(d{:}C;\ j)$:
refines $A(c{:}C = d;\ i = j)$
$\wedge\ j \leq c.y$
$\rightarrow\ c.y' = c.y + j.$

As new conjuncts can be added to enabling condition, *liveness properties* ('something good will eventually happen') are not guaranteed to be preserved by construction. For them, proof obligations are obtained, which can be checked either informally or by a formal proof.

Composition of existing DisCo systems is a refinement of all the component systems, with a straightforward interpretation in terms of layers.

The underlying closed-system philosophy offers a convenient interpretation of layer-based specification. Layers that have been defined form a universe. This universe can be extended by providing new layers, but these new layers cannot assign new values to the variables defined in the layers that already exist. Thus, each layer that defines data must introduce operations needed for modifying its data. Operations may, however, assign nondeterministic values, if the variables that determine the assigned values are not available in the existing layers.

2.3 Inheritance

In DisCo, class refinement is a special form of inheritance. When extending a class, a new class is created, that is a subclass of the original class. In a normal refinement we then implicitly state that no instances of the original class exist outside the extension. If this is undesirable, inheritance can be used explicitly. A class D derived from base class C can be introduced in the format

class $D = C + \{\ldots\}$.

Multiple inheritance is also allowed.

The adoption of inheritance implies a requirement to be able to specialize actions for different kinds of participants. For this purpose, an action B that specializes action A for participants in subclass D can be given as

$B(d{:}D;\ j)$:
refines $A(c{:}C = d;\ i = j)$ **for** $c \in D$.

This results in two actions. Action $B(d{:}D;\ j)$ is available for objects in class D, and $A(c{:}C;\ i)$ is to be used by instances of class C that do not belong to class D.

The closed-system interpretation of inheritance does not differ from that of standard refinements.

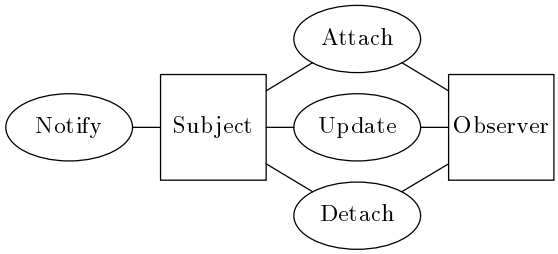

Figure 1: An illustration of the Observer pattern

3 FORMALIZING A DESIGN PATTERN

This section discusses how design patterns can be formalized in terms of DisCo. The focus is on the use of high-level abstractions of cooperation.

As an example, *Observer* pattern given in [7] is formalized. Informally, this pattern can be described as follows. We have subjects and observers. Each subject is a container of data whose contents can be modified, and each observer is an object that can be interested in the contents of a subject. The pattern describes how subjects and observers are connected with each other, and how they communicate in order to preserve data consistency. The pattern is illustrated in Figure 1. Intuitively, the characteristic property of the pattern is that whenever an observer receives data, the subject that the observer is attached to contains the same data.

3.1 Classes and Relations

Based on the description of the pattern, it is relatively obvious that classes whose instances represent subjects and observers are needed. Each such object may have an internal state, yielding

class $Subject = \{Data\}$,

class $Observer = \{Data\}$.

In the formalization, *Data* is included in both classes to model the temporal behavior related to the contents of the associated objects. Obviously, by omitting the data, an abstraction of this specification is obtained. However, as the role of the data is vital in *Observer* pattern, its presence in the formalization is justified.

Two instances of the above classes are associated with each other whenever an observer is interested in the contents of a subject. This relation, representing an attachment of an observer to a subject, can be formalized as

relation $(0..1){\cdot}Attached{\cdot}(*)$: $Subject \times Observer$.

In addition to the attached observers, each subject needs to know to which observers its contents have been

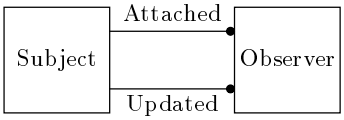

Figure 2: Class diagram of Observer pattern

delivered since the latest modification. Thus, subjects are associated with the observers that have already been updated. This yields the following relation:

relation $(0..1)\cdot Updated\cdot(*)$: $Subject \times Observer$.

A diagram representing classes and their relations is given in Figure 2.

3.2 Actions

In the pattern, an observer can become interested in the contents of a subject, and may also cancel this interest. In the formalization, actions *Attach* and *Detach* are used for modeling arousal and cancelation of interest, respectively. Action *Attach* sets *Attached* relation between the subject and the observer that are involved. Action *Detach* clears this relation, and ensures that possible *Updated* relation between these objects is cleared, too. These actions are formalized as follows:

$$\begin{aligned} &Attach(s{:}Subject;\ o{:}Observer):\\ &\quad \neg s\cdot Attached\cdot o\\ &\rightarrow\ s\cdot Attached'\cdot o, \end{aligned}$$

$$\begin{aligned} &Detach(s{:}Subject;\ o{:}Observer):\\ &\quad s\cdot Attached\cdot o\\ &\rightarrow \neg s\cdot Attached'\cdot o\\ &\wedge \neg s\cdot Updated'\cdot o. \end{aligned}$$

The formalism requires no initiators for actions. Instead, it is a design decision to define who is responsible for initiating exections.

Action *Notify* denotes that the contents of a subject have been modified. At this level of abstraction this is interpreted as a need to update all observers that are attached to the subject. Thus, upon executing *Notify*, the subject must no longer be associated with any observer by *Updated* relation. This results in the following action:

$$\begin{aligned} &Notify(s{:}Subject,\ d):\\ &\rightarrow \neg s\cdot Updated'\cdot \textbf{class}\ Observer\\ &\wedge\ s.Data' = d, \end{aligned}$$

where parameter d models the new value, set upon notification, and **class** *Observer* denotes all instances of class *Observer*. As no restrictions are imposed on the value of parameter d, its value is nondeterministically selected at this level of abstraction.

Action *Update* represents a transmission of modified data from a subject to an observer. Thus, it sets *Updated* relation for the subject and the observer that participate in the action, yielding

$$\begin{aligned} &Update(s{:}Subject;\ o^*{:}Observer;\ d):\\ &\quad s\cdot Attached\cdot o\\ &\wedge\ \neg s.Updated.o\\ &\wedge\ d = s.Data\\ &\rightarrow\ s\cdot Updated'\cdot o\\ &\wedge\ o.Data' = d. \end{aligned}$$

Marking participant o with an asterisk denotes a *fairness requirement*, stating that if an object could repeatedly take this role in this action, the action will be executed for the object. Such requirements are essential for liveness properties.

4 COMBINING PATTERNS

In this section, we combine patterns to form specifications of more complex systems. The emphasis is on how cooperation is refined to contain behaviors of patterns, each of which is introduced in a behavioral layer of its own. Of particular interest is that in many cases operations of different patterns need to take place synchronously. Straightforward formalization of the temporal behavior results, because implementation-dependent protocols or method invocations are omitted at this level of abstraction.

In [7], *Observer* pattern is combined with *Mediator* pattern, resulting in a special version of the original patterns, where the information between subjects and observers is transmitted via a manager. This combination is referred to as *Managed Observer*. In the following, *Managed Observer* is formalized by using DisCo refinements, starting from the formalizations of the underlying patterns. Due to the nature of DisCo refinements, the resulting system is a correct combination of the patterns in the sense that the characteristic properties of both underlying patterns are preserved.

4.1 Mediator Pattern

Mediator pattern given in [7] can be informally described as follows. There is a set of colleagues that can connect to a mediator. Connected colleagues can communicate with each other by putting messages into the mediator, which then lets the colleagues get the messages addressed to them. When a connection is no longer needed, colleagues may disconnect from the mediator. Figure 3 illustrates the structure of the pattern. Intuitively, the characteristic property of this pattern is that whenever a colleague receives a message, the message has been explicitly sent to it by a colleague who

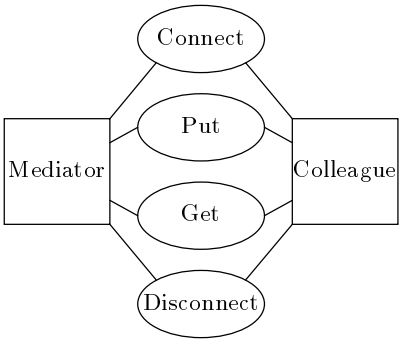

Figure 3: An illustration of the Mediator pattern

still is connected to the mediator.

Object identity is needed for each colleague, enabling a particular colleague to act as the sender or a receiver of a message. This is formalized as an empty class,

class *Colleague*.

Whenever colleagues communicate, messages are delivered by using mailboxes, formalized as follows:

class $Mbox = \{Message\}$.

The sender and the receivers of a message are modeled as relations that involve colleagues and a mailbox that contains the message. This results in the following relations:

relation $(0..1)\cdot Sender\cdot(0..1)$: $Mbox \times Colleague$,

relation $(*)\cdot Receiver\cdot(*)$: $Mbox \times Colleague$.

The mediator that supports the communication consists of a set of mailboxes, resulting in the following class definition:

class $Mediator = \{\ \{Boxes\} : Mbox\}$,

where $\{Boxes\}$ denotes a set of instances of class *Mbox*.

An object of *Mediator* class is associated with those instances of *Colleague* class that are connected to it. This is defined as a relation,

relation $(0..1)\cdot Connected\cdot(*)$:
$Mediator \times Colleague$.

The above definitions imply a class diagram illustrated in Figure 4.

From the behavioral point of view, colleagues need be able to connect to and disconnect from mediators.

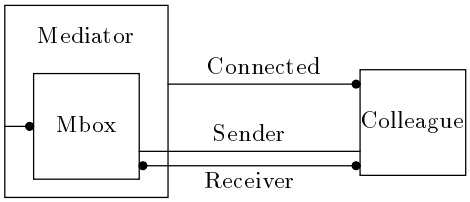

Figure 4: Class diagram of Mediator pattern

When connecting, the associated colleague is given a mailbox that it can use for sending messages, denoted by *Sender* relation. When disconnecting, *Sender* relation is cleared, and messages involving the colleague are canceled. Such actions are formalized as follows:

$$
\begin{aligned}
&Connect(m{:}Mediator;\ mb{:}Mbox;\ c{:}Colleague):\\
&\qquad \neg m\cdot Connected\cdot c\\
&\quad \wedge\ \neg mb\cdot Sender\cdot \textbf{class}\ Colleague\\
&\quad \wedge\ mb \in m.Boxes\\
&\rightarrow\ m\cdot Connected'\cdot c\\
&\quad \wedge\ mb\cdot Sender'\cdot c,
\end{aligned}
$$

$$
\begin{aligned}
&Disconnect(m{:}Mediator;\ mb{:}Mbox;\ c{:}Colleague):\\
&\qquad m\cdot Connected\cdot c\\
&\quad \wedge\ mb\cdot Sender\cdot c\\
&\quad \wedge\ mb \in m.Boxes\\
&\rightarrow\ \neg m\cdot Connected'\cdot c\\
&\wedge \neg mb\cdot Sender'\cdot c\\
&\quad \wedge\ \neg\textbf{class}\ Mbox\cdot Receiver'\cdot c.
\end{aligned}
$$

When a set of colleagues is connected to the same mediator, they can send messages to each other. This message transmission is supported by the associated mediator. Colleagues communicate by putting messages into the mediator, and by getting the messages addressed to them from it, resulting in the following formalizations:

$$
\begin{aligned}
&Put(m{:}Mediator;\\
&\qquad mb{:}Mbox;\\
&\qquad from,\ \{to\}{:}Colleague;\\
&\qquad d):\\
&\qquad m\cdot Connected\cdot from\\
&\quad \wedge\ m\cdot Connected\cdot to\\
&\quad \wedge\ mb\cdot Sender\cdot from\\
&\quad \wedge\ mb \in m.Boxes\\
&\rightarrow\ mb\cdot Receiver'\cdot to\\
&\quad \wedge\ \neg mb\cdot Receiver'\cdot(\textbf{class}\ Colleague\text{ - }to)\\
&\quad \wedge\ mb.Message' = d,
\end{aligned}
$$

$$
\begin{aligned}
&Get(m{:}Mediator;\ mb{:}Mbox;\ c^*{:}Colleague;\ d):\\
&\qquad m\cdot Connected\cdot c\\
&\quad \wedge\ mb\cdot Receiver\cdot c\\
&\quad \wedge\ mb \in m.Boxes\\
&\quad \wedge\ mb.Message = d\\
&\rightarrow\ \neg mb\cdot Receiver'\cdot c,
\end{aligned}
$$

where $\{to\}$ denotes a set of suitable receivers, and expression (**class** *Colleague* - *to*) refers to those instances

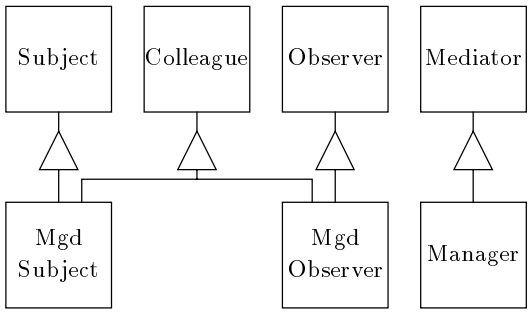

Figure 5: Inheritance hierarchy in the combination

of class *Colleague* that do not belong to set *to*. A fairness requirement is given to ensure that all colleagues will eventually receive messages addressed to them.

This formalization restricts the number of messages being sent by each colleague to at most one, because whenever a colleague puts a new message into the mediator, its previous message is overwritten. A message can, however, be addressed to several receivers.

4.2 Combination of Patterns

Above, patterns needed for constructing *Managed Observer* have been formalized. In the following we form their combination, which defines how the internal state of a subject is transmitted to the attached observers by using messages supported by the mediator. In the combination, the behavior of both underlying patterns is guaranteed to be preserved because only property-preserving refinements are used. Thus, by restricting to observations obtained from a layer that formalizes a single pattern, the behavior of the combination can be projected into the pattern.

In terms of *Observer* pattern, subjects and observers are refined into a form where they are managed. The manager of this data transmission behaves like the mediator in *Mediator* pattern, yielding

class $Manager = Mediator.$

Managed subjects and observers are subjects and observers whose data transmissions take place as defined by inter-colleague communication in *Mediator* pattern. This results in the following class definitions:

class $MgdSubject = Subject + Colleague,$

class $MgdObserver = Observer + Colleague.$

These definitions result in the class hierarchy illustrated in Figure 5.

Due to the use of inheritance, actions need to be specialized for different types of participants. Actions that connect or disconnect subjects and managers are similar to those defined by the mediator pattern, with the exception that a subject cannot be disconnected if an observer is attached to it through the associated manager. Thus, the following actions result:

$$\begin{aligned} &MgdConnect(mgr{:}Manager;\\ &\qquad mb{:}Mbox;\\ &\qquad ms{:}MgdSubject):\\ &\textbf{refines}\ Connect(m{:}Mediator = mgr;\\ &\qquad mb{:}Mbox = mb;\\ &\qquad c{:}Colleague = ms)\\ &\qquad \textbf{for}\ m \in Manager,\ c \in MgdSubject, \end{aligned}$$

$$\begin{aligned} &MgdDisonnect(mgr{:}Manager;\\ &\qquad mb{:}Mbox;\\ &\qquad ms{:}MgdSubject;):\\ &\textbf{refines}\ Disconnect(m{:}Mediator = mgr;\\ &\qquad mb{:}Mbox = mb;\\ &\qquad c{:}Colleague = ms)\\ &\qquad \textbf{for}\ m \in Manager,\ c \in MgdSubject\\ &\qquad\quad \wedge \neg ms{\cdot}Attached{\cdot}\textbf{class}\ MgdObserver. \end{aligned}$$

The resulting actions formalize an important aspect. After this definition, only action *MgdConnect* can connect instances of class *MgdSubject* to *Manager*. Action *Connect* is restricted to use objects that do not belong to either of these subclasses. A similar difference exists between actions *MgdDisconnect* and *Disconnect*.

For observers, the situation is more complicated. We can no longer simply attach pairs of subjects and observers, or connect colleagues and mediators. Instead, the task is to register triples that create an association between a managed subject, a manager, and a managed object. This results in an action that represents action *Attach* for the managed subject and the managed observer, and action *Connect* for the observer and the manager. In addition, the associated manager and the subject need to be connected with each other, as otherwise the manager has no access to the contents of the subject. This results in the following action:

$$\begin{aligned} &Register(ms{:}MgdSubject;\\ &\qquad mo{:}MgdObserver;\\ &\qquad mb{:}Mbox;\\ &\qquad mgr{:}Manager):\\ &\textbf{refines}\ Attach(s{:}Subject = ms;\\ &\qquad o{:}Observer = mo)\\ &\qquad \textbf{for}\ s \in MgdSubject,\ o \in MgdObserver\\ &\&\textbf{refines}\ Connect(m{:}Mediator = mgr;\\ &\qquad mb{:}Mbox = mb;\\ &\qquad c{:}Colleague = mo)\\ &\qquad \textbf{for}\ m \in Manager,\ c \in MgdObserver\\ &\qquad\quad \wedge\ mgr{\cdot}Connected{\cdot}ms. \end{aligned}$$

Here, one action refines both *Attach* and *Connect* actions. In DisCo, this way of combining actions is a key

aspect in multiple inheritance. From the viewpoint of the underlying patterns, the execution of *Attach* and *Connect* is synchronized for the objects in the derived classes.

Similar refinement is given for canceling a registration, yielding

Release(*ms*:*MgdSubject*;
mo:*MgdObserver*;
mb:*Mbox*;
mgr:*Manager*):
refines *Detach*(*s*:*Subject* = *ms*;
o:*Observer* = *mo*)
for $s \in MgdSubject,\ o \in MgdObserver$
&**refines** *Disconnect*(*m*:*Mediator* = *mgr*;
mb:*Mbox* = *mb*;
c:*Colleague* = *mo*)
for $m \in Manager,\ c \in MgdObserver.$

In the *Observer* pattern, a change of the value of a subject causes only the corresponding modification of the internal state. For the managed version, however, the associated manager need to be informed about the change as well, as it must be able to cancel obsolete messages from being dispatched to the associated observers. No explicit cancel action exists, but effectively the same is accomplished by executing *Put* action, defined in *Mediator* pattern, with an empty set of receivers. This can be formalized as follows:

Modify(*ms*:*MgdSubject*;
mb:*Mbox*;
mgr:*Manager*;
d):
refines *Notify*(*s*:*Subject* = *ms*; *d* = *d*)
for $s \in MgdSubject$
&**refines** *Put*(*m*:*Mediator* = *mgr*;
mb:*Mbox* = *mb*;
from:*Colleague* = *ms*;
to:*Colleague* = {};
d = *d*)
for $m \in Manager,\ from \in MgdSubject.$

Notice that it is essential that obsolete data is never delivered to observers, because such a specification is not a correct refinement of *Observer* pattern. If the method would not enforce such issues, they might be included in an implementation unnoticed, possibly resulting in misbehaviors.

When the state of a managed subject is modified, *Observer* pattern requires that all attached observers will be informed about the new state of the subject. An action is therefore needed that transmits the current state of a subject to the manager, which is responsible for dispatching this data to those observers that are attached to the subject. Thus, action *Transmit* refines *Put* action in a manner where the managed subject is the sender, and (some of) the receivers attached to it are the receivers[1]. This results in the following formalization:

Transmit(*ms*:*MgdSubject*;
$\{mo\}^*$:*MgdObserver*;
mb:*Mbox*;
mgr:*Manager*;
d):
refines *Put*(*m*:*Mediator* = *mgr*;
mb:*Mbox* = *mb*;
from:*Colleague* = *ms*;
to:*Colleague* = *mo*;
d = *d*)
for $m \in Manager,$
$from \in MgdSubject,$
$to \in MgdObserver$
$\wedge\ d = ms.Data$
$\wedge\ ms{\cdot}Attached{\cdot}mo$
$\wedge\neg ms{\cdot}Updated{\cdot}mo.$

Liveness properties are preserved by requiring fairness with respect to possible observers, denoted by an asterisk. A tempting implementation, where the subject transmits its new value immediately upon modification, is obtained by synchronizing the exection of action *Transmit* with *Modify* operation.

In *Observer* pattern, when an observer notices that the value of a subject it is attached to has been updated, the observer modifies its contents accordingly. In the combination of patterns, the observer can notice such a change only when it gets a message from a manager. The following formalization therefore results:

Dispatch(*ms*:*MgdSubject*;
mo^*:*MgdObserver*;
mb:*Mbox*;
mgr:*Manager*;
d):
refines *Update*(*s*:*Subject* = *ms*;
o:*Observer* = *mo*,
d = *d*)
for $s \in MgdSubject,\ o \in MgdObserver$
&**refines** *Get*(*m*:*Mediator* = *mgr*;
mb:*Mbox* = *mb*;
c:*Colleague* = *mo*;
d = *d*)
for $m \in Manager,\ c \in MgdObserver$
$\wedge\ mb{\cdot}Sender{\cdot}ms.$

A fairness requirement is again needed for satisfying the liveness properties of the original patterns. Parameter *d*

[1] This formalization does not define any update strategy. Therefore, the specification allows implementations where the new data is broadcasted to all the attached observers as well as ones where only one receiver per message is possible.

formalizes an important aspect, requiring that the values contained in the message and the subject need to be identical. If it can be proved that the values are always equal, the check may be ignored, possibly allowing implementations where synchronizations with the subject are omitted when dispatching the data. Such proofs, liberating the implementation from unnecessary synchronizations, are essential when deriving a specification towards an implementation [15].

This completes the specification, demonstrating that the characteristic properties of both underlying patterns are satisfied by the combination. We have thus shown the ability to combine patterns in a manner that produces specifications which are refinements of all underlying patterns.

5 FORMALIZING AN INSTANTIATION

The focus of this section is on showing how to include application-specific parts in the specification. As an example, *Managed Observer* defined above is instantiated, using statistics as concrete data. Although this data refinement is relatively simple, it demonstrates that the mechanisms used for combining patterns are applicable to the specification of complete systems that utilize patterns as their skeletons. The approach is therefore uniform with respect to patterns and systems built by utilizing them. In addition, the use of layers gives the specification a structure where each individual pattern can be identified even when the specification is completed, supporting the interpretation where patterns are used as building blocks for more complex systems.

5.1 Classes

Concrete subjects and observers are based on their managed counterparts described by *Managed Observer*,

class $StatSubject = MgdSubject$,

class $StatObserver = MgdObserver$.

As we wish to refine the specification to a level where concrete data exists, a representation for the data is required. The relation between the actual data and the abstraction of it, referred to as *Data* in the original pattern, must also be formalized. Assuming that pair $\langle a, b\rangle$ denotes concrete statistics data included in each instance of the new subclasses, we require it to be invariably true that

$\forall ss \in StatSubject::\ ss.Data = \langle ss.a,\ ss.b\rangle$,

$\forall so \in StatObserver::\ so.Data = \langle so.a,\ so.b\rangle$.

In other words, variable *Data* is given a structure that corresponds to statistics data.

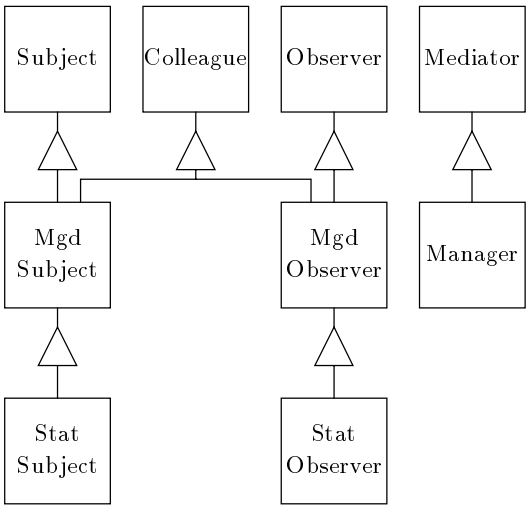

Figure 6: Inheritance in the concrete system

These class definitions form a class hierarchy illustrated in Figure 6.

5.2 Actions

As new classes were derived, actions reflecting the operational difference between the instances of the original and the derived classes need to be defined. Subjects can be connected and disconnected as before, but only statistics observers are registered to observe statistics subjects. This results in the following formalization:

RegisterStat(*ss*:*StatSubject*;
so:*StatObserver*;
mb:*Mbox*;
mgr:*Manager*):
refines *Register*(*ms*:*MgdSubject* = *ss*;
mo:*MgdObserver* = *so*;
mb:*Mbox* = *mb*;
mgr:*Manager* = *mgr*)
for $ms \in StatSubject,\ so \in StatObserver$.

In order to preserve symmetry, we also specialize action *Release* in a similar manner, although this specialization is not absolutely necessary. The following formalization results:

ReleaseStat(*ss*:*StatSubject*;
so:*StatObserver*;
mb:*Mbox*;
mgr:*Manager*):
refines *Release*(*ms*:*MgdSubject* = *ss*;
mo:*MgdObserver* = *so*;
mb:*Mbox* = *mb*;
mgr:*Manager* = *sm*)
for $ms \in StatSubject,\ mo \in StatObserver$.

When data is modified, actual statistics values are needed. Action *Modify* is therefore specialized to handle real data, which is added to the action by using parameters. In addition, the actual data is required to be more precise versions of the original data, thus ensuring that the data refinement is correct. In the formalization, statistics data is represented by using parameters a and b. The action is formalized as follows:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{ModifyStat}(\mathit{ss}{:}\mathit{StatSubject};} \\
& \mathit{mb}{:}\mathit{Mbox}; \\
& \mathit{mgr}{:}\mathit{Manager}; \\
& a,\ b): \\
\textbf{refines} & \mathit{Modify}(\mathit{ms}{:}\mathit{MgdSubject} = \mathit{ss}; \\
& \mathit{mb}{:}\mathit{Mbox} = \mathit{mb}; \\
& \mathit{mgr}{:}\mathit{Manager} = \mathit{mgr}; \\
& d = \langle a,\ b \rangle) \\
\textbf{for} & \mathit{ms} \in \mathit{StatSubject}.
\end{array}
$$

Data refinement only concerns those parts of the specification that deal with the actual meaning of the data. As action *Transmit* only sends the data, without paying any attention to its contents, the action need not be specialized for statistics. The practical meaning of this is that there is no need to give separate implementation for transmitting statistics data. Instead, all data is transmitted in a similar manner. This decision is intuitively well-justified, as it seems natural to hide any concerns regarding the meaning of the data from the manager.

When an observer receives data, the data may be plain data sent by a subject outside class *StatSubject*, or statistics sent by an instance of this class. Due to the specialized registration, observers in class *StatObserver* only obtain statistics, and the other observers are restricted to receive plain data. Action *Dispatch* can therefore be specialized for statistics. In addition to formalizing data refinement, specialization of action *Dispatch* must also take into account associated fairness requirements. Such a specialization can be formalized as follows:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{DispatchStat}(\mathit{ss}{:}\mathit{StatSubject};} \\
& \mathit{so}^{*}{:}\mathit{StatObserver}; \\
& \mathit{mb}{:}\mathit{Mbox}; \\
& \mathit{mgr}{:}\mathit{Manager}; \\
& a,\ b): \\
\textbf{refines} & \mathit{Dispatch}(\mathit{ms}{:}\mathit{MgdSubject} = \mathit{ss}; \\
& \mathit{mo}{:}\mathit{MgdObserver} = \mathit{so}; \\
& \mathit{mb}{:}\mathit{Mbox} = \mathit{mb}; \\
& \mathit{mgr}{:}\mathit{Manager} = \mathit{mgr}; \\
& d = \langle a,\ b \rangle) \\
\textbf{for} & \mathit{ms} \in \mathit{StatSubject},\ \mathit{mo} \in \mathit{StatObserver}.
\end{array}
$$

Although the specification is now instantiated, further refinements are still possible. Moreover, these refinements can address the platform as well as application-specific aspects. For example, additional refinements could introduce a message transmission strategy based on priorities, or specialize *StatObserver* to introduce different kinds of observers, using data for different purposes.

6 CONCLUSIONS

When a pattern is formalized, it is little more than a formal specification that is generic with respect to aspects that are not essential to be fixed. The main advantage of the formalization of a pattern, or any formalization, for that matter, is that there is no ambiguity. This absense of ambiguity allows us to rigorously address the temporal behaviors of patterns. Reasoning enabled by this rigorousness can be made either by using a theorem prover [11] or by hand [10].

Formalization of patterns can be eased by using well-defined units of modularity. Above, each pattern was formalized as a behavioral layer, introducing slices of objects that resemble program slices [17]. Therefore, underlying patterns can be easily identified in a completed specification, enabling interpretations where patterns are naturally used as building blocks for specifications of more complex systems. Moreover, use of layers enable projections of behaviors defined by complete specifications to individual patterns. When formalizing combinations of design patterns, an essential element is multiple inheritance. Without it, the combination of patterns resulting in *Managed Observer* would have been difficult, or impossible to formalize in a natural manner.

Complicated communication between objects has been simplified by using an abstract notion of atomic actions. We have therefore been able to concentrate on the bare essentials of the systems, placing the emphasis on inter-object cooperation instead of invocations of single-object methods. This raises the level of abstraction above programming-level abstractions [13]. Standard refinements can be used to derive abstract cooperation into directly implementable communication [15].

Property-preserving refinements supported by the DisCo method provide a sophisticated way to combine rigorous and pattern-oriented software development. An implication of property-preserving refinements is that the properties of a pattern can be validated and verified at pattern level, and when using the pattern, refinement steps enforce that the pattern is correctly included in the resulting specification. This use of property-preserving refinements is facilitated by composing the specification in terms of atomic actions only, omitting implicit causality between them. Nondeterminism that facilitates this is therefore of crucial importance for achieving genericity.

In brief, the DisCo method offers object-oriented modeling capabilities that can be used for developing specifications at a high level of abstraction, as well as well-defined semantics that enable rigorous reasoning. Due to appropriate units of modularity and adequate level of abstraction, formalization of design patterns and specification of systems obtained by utilizing them is rigorous but practical.

REFERENCES

[1] P. S. C. Alencar, D. D. Cowan, C. J. P. Lucena. A formal approach to architectural design patterns. *FME'96: Industrial Benefit and Advances in Formal Methods* (Eds. M.-C. Gaudel, J. Woodcock), 576–594, Springer-Verlag LNCS 1051, 1996.

[2] R. J. R. Back, R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing 3*, 73–87, May 1989.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns.* John Wiley & Sons, 1996.

[4] K. M. Chandy, J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[5] J. S. Fitzgerald, P. G. Larsen, T. M. Brookes, and M. A. Green. Developing a security-critical system using formal and conventional methods. *Applications of Formal Methods*(Eds. M. Hinchey, J. Bowen), 333–356, Prentice-Hall, 1995.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming* (Ed. M. Nierstrasz), 406–431, Springer-Verlag LNCS 707, 1993.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] D. Garlan. The role of formal frameworks. *ACM SIGSOFT Software Engineering Notes 15*, 4, 42–44, September 1990.

[9] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, 63–71, IEEE Computer Society Press, 1990.

[10] P. Kellomäki. *Analysis of a Stabilizing Protocol.* Licentiate thesis, Tampere University of Technology, 1994.

[11] P. Kellomäki. Verification of reactive systems using DisCo and PVS. *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, (Eds. J. Fitzgerald, C. B. Jones, P. Lucas), 589–604, Springer-Verlag LNCS 1313, 1997.

[12] R. Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. *Object-Oriented Behavioral Specifications* (Eds. H. Kilov and W. Harvey), 101–120, Kluwer, 1996.

[13] R. Kurki-Suonio, T. Mikkonen. Liberating object-oriented modeling from programming-level abstractions. *Workshop on Precise Semantics for Object-Oriented Models (ECOOP'97)*, to be published by Springer-Verlag.

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

[15] T. Mikkonen. *Implementation of Reactive Systems Based on Closed-System Specifications.* Licentiate thesis, Tampere University of Technology, 1995.

[16] K. Systä. A graphical tool for specification of reactive systems. In *Proceedings of Euromicro'91 Workshop on Real-time Systems*, 12–19, Paris, France, June 12–14, 1991.

[17] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No 4, 352–357, 1984.