# (Behavioural) Design Patterns as Composition Operators

Kung-Kiu Lau, Ioannis Ntalamagkas, Cuong M. Tran, and Tauseef Rana

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu,intalamagkas,ctran,ranat}@cs.manchester.ac.uk

**Abstract.** Design patterns are typically defined informally, albeit in a standard format, and have to be programmed by the software designer into each new application. Thus although patterns support solution reuse, in practice this does not translate into code reuse. In this paper we argue that to achieve code reuse, patterns should be defined and used in the context of software component models. We show how in such a model, behavioural patterns can be defined as composition operators which can be stored in a repository, alongside components, thus enabling code reuse.

## 1 Introduction

Design patterns [5], as generic reusable solutions to commonly occurring problems, are one of the most significant advances in software engineering to date, and have become indispensable tools for object-oriented software design. However, a pattern is typically only defined informally, using a standard format containing sections for pattern name, intent, motivation, structure, participants, etc. To use a pattern for an application, a programmer has to understand the description of the pattern and then work out how to program the pattern into the application. Although patterns are supposed to encourage code reuse (by way of solution reuse), in practice such reuse does not happen, since the programmer has to program the chosen pattern practically from scratch for each application.

In this paper we argue that to really achieve code reuse, patterns should be defined and used in the context of *software component models* [8,19]. Moreover, patterns should be formal entities in their own right, so that they are units with their own identity that can be composed with specific components to form a solution to a specific problem. In other words, patterns should be explicitly defined *composition operators* (in a component model) that can be used to compose components. As composition operators, patterns would be like functions with generic parameters, and as such would be reusable with different sets of components for different problems. Furthermore, the semantics of a pattern can be defined formally, and then embodied in the corresponding composition operator, so that choosing a pattern can be done on the basis of formal semantics, rather than informal description, as is current practice.

In our work on software component models [18,15], we have defined (a component model with) explicit composition operators. Such operators can themselves be composed into composite operators [13]. In this paper, we show how we define composition operators, in particular composite ones, and how (some) behavioural patterns can be defined as such operators. We define the *Chain of Responsibility* (CoR) pattern as a basic composition operator, the *Observer* pattern as a composite composition operator, and a composite pattern composed from CoR and Observer as another composite operator composed from the two former operators. We also show an implementation in which patterns are stored in a repository, alongside components, thus enabling code reuse.

## 2 Related Work

Design patterns have been formalised by using some formalisation of objects and their relationships. For example, in [24] patterns are defined by using the object-oriented specification language DISCO [10]. These approaches basically take the informal description of a pattern, as given in e.g. [5], and re-write it in a formal manner. However, they do not define patterns as operators that can be applied to generic parameters. Consequently, the formalisation provides just another definition of patterns, and the programmer still has to program a chosen pattern from scratch for each application. Therefore there is no code reuse.

Composite design patterns [33] and techniques for composing patterns have also been investigated. For example, the composite patterns active bridge, bureaucracy and model-view-controller were proposed in [27] Composition techniques can be classified as (i) stringing or (ii) overlapping [35,7]. In stringing, patterns are glued together; in overlapping, a participant in one pattern also participates in another pattern at the same time. In these techniques, composition is constrained by relationships between roles. For example, design patterns that are architectural fragments are composed by merging roles in [1] using superimposition. However, these techniques are defined informally, and are applied in an *ad hoc* manner. Therefore, they do not support systematic code reuse.

To achieve code reuse in a more direct manner, there has been research into componentising patterns, by implementing packages for patterns that programmers can use to program patterns for different applications. For example, [23] shows that two thirds of the common patterns like Visitor can be 'componentised' in this way. Patterns are implemented as Eiffel packages that require other Eiffel packages. However, in a pattern package, the roles of and the constraints on the participant objects are not (cannot be) specified. As a result, a package does not truly represent a pattern. Although some code reuse is achieved by the use of skeleton code provided by packages, most of the coding effort remains, in partcular for code that defines the participants' roles and constraints.

Component composition patterns were identified in [34] to define domain-specific communication patterns using modified sequence diagrams. Component roles are used to restrict the behaviour of the participating components, but in terms of their interface behaviour. The focus of this work is on the definition of domain-specific communication patterns and not on generic software patterns, and pattern composition is undefined.

## 3 Our Approach

We believe that true code reuse can be achieved by defining design patterns in the context of a properly defined software component model. Such a model defines what components are, and mechanisms for composing them.

A generic view of a component is a composition unit with required services and provided services. Following UML2.0 [25], this is expressed as a box with lollipops (provided services) and sockets (required services), as shown in Fig.1(a). In current software component models [19], components are either *objects* or *architectural units*. Exemplars of these models are EJB [4] and ADLs (architecture description languages) [21] respectively. An object normally does not have an interface, i.e. it does not specify its required services or its provided services (methods), but in component models like JavaBeans [29] and EJB, beans are objects with an interface showing its provided methods but usually not its required services (Fig.1(b)). Architectural units have input ports as required services and output ports as provided services (Fig.1(c).) Therefore, objects and architectural units can both be represented as components of the form in Fig.1(a).

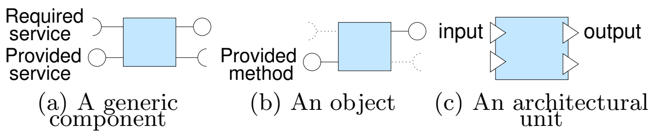


**Fig. 1.** Components

Objects and architectural units are composed by *connection* (Fig.2), whereby matching provided and required services are connected by assembly connectors. In order to get a required service from another object, an object calls the appropriate method in that object. Thus objects are connected by method delegation, i.e. by direct message passing. For architectural units, connectors between ports provide communication channels for indirect message passing.[1]

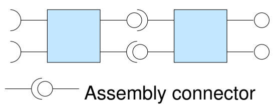


**Fig. 2.** Connection

We have defined a component model [18,15] in which composition operators are explicitly defined entities with their own identities. In our model, components are *encapsulated*: they encapsulate control, data as well as computation, as in 'enclosure in a capsule'. Our components have no external dependencies, and can therefore be depicted as shown in Fig.3(a), with just a lollipop, and no socket. There are two basic types of components: (i) *atomic* and (ii) *composite.*

[1] In [30] object delegation and architectural unit composition are differentiated.

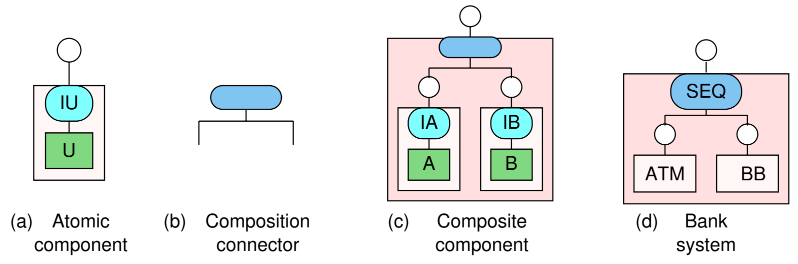


**Fig. 3.** Our component model

Fig 3(a) shows an atomic component. This consists of a *computation* unit (U) and an *invocation connector* (IU). A computation unit contains a set of methods which do not invoke methods in the computation units of other components; it therefore encapsulates computation. An invocation connector passes control (and input parameters) received from outside the component to the computation unit to invoke a chosen method, and after the execution of method passes control (and results) back to whence it came, outside the component. It therefore encapsulates control.

A composite component is built from atomic components by using a *composition connector*. Fig.3(b) shows a composition connector. This encapsulates a control structure, e.g. sequencing, branching, or looping, that connects the subcomponents to the interface of the composite component (Fig.3(c)). Since the atomic components encapsulate computation and control, so does the composite component. Our components therefore encapsulate control (and computation)[2] at every level of composition. Note that we have emphasised the significance of control encapsulation in [16].

Our components can be active or passive. Active components have their own threads and execute autonomously, whereas passive components only execute when invoked by an external agent. In typical software applications, a system consists of a 'main' component that initiates control in the system, as well as components that provide services when invoked, either by the 'main' component or by the other components. The 'main' component is active, while the other components are passive. For simplicity, in this paper we focus on passive components in our model; composition of active components is much more involved, by comparison (see [14]). In our model, passive components receive control from, and return it, to connectors. In a system, control flow starts from the top-level (composition) connector.

Fig.3(d) shows a simplified bank system with two components *ATM* and *BB* (bank branch), composed by a sequencer composition connector *SEQ*. Control starts when the customer keys in his PIN and the operation he wishes to carry out. The connector *SEQ* passes control to *ATM*, which checks the customer's PIN; then it passes control to *BB*, which gets hold of the customer

[2] As well as data [17].

account details and performs the requested operation. Control then passes back to the customer.

In summary, composition in our model is hierarchical: components can be 're-cursively' composed into larger composites, as can be seen in Figs. 3(c) and 3(d).

## 4 Composition Operators

In [18] we defined the composition operators informally, and in [15] we defined them in terms of many-sorted first-order logic. In addition, we defined a catalog of composition operators in [32]. To relate our composition operators to behavioural patterns, in this section we give the formal semantics of composition operators in terms of Coloured Petri nets [11].

First, it is worth emphasising that, as we saw in Section 3, our composition operators are connectors [18] that encapsulate control. Moreover, these operators themselves can be composed to yield composite operators that also encapsulate control. This is illustrated in Fig.4 for one thread of control for the sequencer composition operator.

For brevity, we will refer to composition operators simply as connectors.

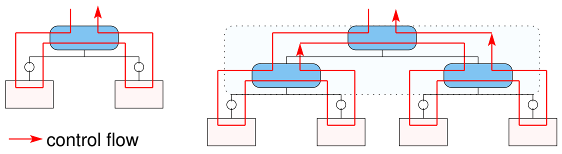


**Fig. 4.** Control encapsulation

### 4.1 Connector Template Nets

We will define our connectors as a special kind of Coloured Petri net [11], which we call a *Connector Template net* (CT-net). A connector in our model is of arbitrary arity and parametricity, and therefore cannot be defined directly using Coloured Petri nets. A Petri net[3] is a set of places (with tokens) and transitions connected by arcs with expressions; and a Coloured Petri net is a Petri net in which the tokens can be of different types (colours).

**Definition 1.** *A* Connector Template net *(CT-net) is a tuple* $(N, Ar, \Sigma, P, T, A, C, G, E, I, In, Out, CP)$*, where:*

*(i)* $N$ *is the unique* name *of the CT-net.*
*(ii)* $Ar$ *is an expression (containing at least one variable) defining the* arity *of the connector.*
*(iii)* $\Sigma$ *is the colour set defining the types used in the CT-net,* $P$*,* $T$*,* $A$ *are disjoint sets of places, transitions and arcs, where* $P$ *and* $T$ *are basic sets, whereas* $A$ *is of type* $P \times T \cup T \times P$*.*

[3] We assume familiarity with Petri nets.

(iv) *$C$ is a function defining the types for each place in $P$, $G$ defines guard expressions for transitions in $T$, $E$ is a function defining the arc expressions in $A$, and $I$ defines the initial marking for each place in $P$.*

(v) *$In$ and $Out$ are distinguished* input *and* output places *of the CT-net, s.t. $\{In, Out\} \subset P \wedge {}^\bullet In = \emptyset \wedge Out^\bullet = \emptyset$, where ${}^\bullet n$ and $n^\bullet$ denotes the* preset *and the* postset *of $n$, i.e. the set of nodes in $P \cup T$ such that there is an arc from a node in the set to $n$, or from $n$ to some node in the set respectively.*

(vi) *CP is the distinguished set of* composition places *of the CT-net, s.t. $CP \subset P \wedge \#CP = Ar \wedge \forall cp \in CP, \#{}^\bullet cp = \#cp^\bullet$, i.e. the cardinality of composition places equals the arity* Ar *of the connector, and the number of input transitions to each composition place equals the number of output transitions of the same place.*

For simplicity, we have defined the arc set $A$ in a CT-net as pairs of nodes (places or transitions), in part (iii) of Definition 1. This introduces the limitation that between each pair of nodes we can define at most one arc, whereas in Coloured Petri nets multiple arcs are allowed.

However, this limitation poses no problems since multiple arcs for a pair of nodes can always be merged into a single arc [11].

Graphically, a CT-net can be depicted as in Fig.5. It has a set of distinguished places: an *input* place *In*, an *output* place *Out*, and a set of *composition* places $CP_1, \ldots, CP_n$, connected to transitions (boxes) in a Coloured Petri net (the dotted box). Each composition place $CP_i$ represents a connector or a component, and has precisely one incoming $in_i$ and one outgoing arc $out_i$.

A CT-net encapsulates control that flows in through its input place, its internal Coloured Petri net, its composition places, back through the internal Coloured Petri net, and finally out through its output place. Control encapsulation in a CT-net in Fig.5 is therefore the same as that defined in Fig.4 for a (composite) connector, for each thread of control.

Concretely we will only use CT-nets with fixed arities defined in a toolkit called CPN Tools [3] for Coloured Petri nets. In these concrete CT-nets, places (and hence tokens) are of type $N \times CID$, where $N$ is the type of natural numbers, and *CID* is the cartesian product of two integer types. *CID* is a case identifier that distinguishes between different (initial) threads corresponding to requests

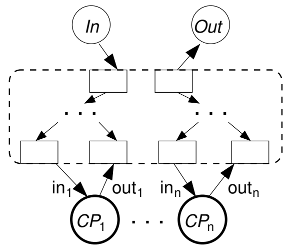


**Fig. 5.** A CT-net

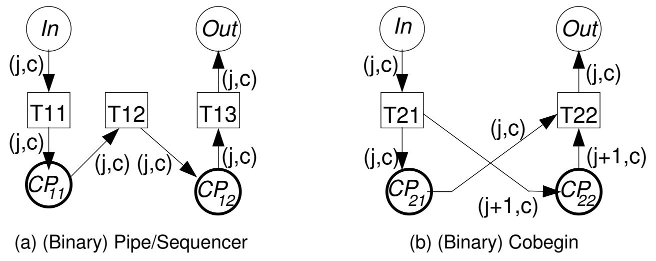


**Fig. 6.** Basic composition operators

by different "users" of a connector, and $N$ is used to identify different sub-threads (see the discussion on the *Cobegin* connector below).

Now we show how connectors in our component model can be defined as CT-nets. We distinguish between *basic* and *composite* connectors.

## 4.2 Basic Composition Operators

Basic connectors in our model are connectors for sequencing, branching and (finite) looping. Fig.6 shows the CT-nets for the *Pipe* connector (for sequencing) and the *Cobegin* connector (for branching)[4]. The *Pipe* connector receives requests and passes them to the components sequentially. The pipe also enables passing results from one component to the next. The CT-net for *Pipe* is the same as that for the *Sequencer* connector. The *Sequencer* is the same as the *Pipe* except it does not pass results. The *Cobegin* connector splits each incoming thread into 2 sub-threads (sharing the same $CID$), that execute concurrently the connected components.

In terms of CT-nets, every basic connector has the following property:

*Property 1.* The control flow of each connector guarantees that each token in the input place will eventually flow to the output place of the connector. In the output place there can only appear the tokens that have previously appeared in the input place, and for each token in the input place there will be exactly one token in the output place.

This property simply states that incoming control threads do not vanish during connector execution, and only these threads return as a result of the connector execution. This property can be trivially proved for the basic connectors, and it must be preserved during connector composition.

## 4.3 Composite Composition Operators

Connectors can be composed via their input, output and composition places: a composition replaces a composition place by its matching input and output places, and re-directing its in-arc to the input place and its out-arc from the output place respectively. This is illustrated in Fig.7, where the two CT-nets are

[4] For simplicity, we only consider binary connectors.

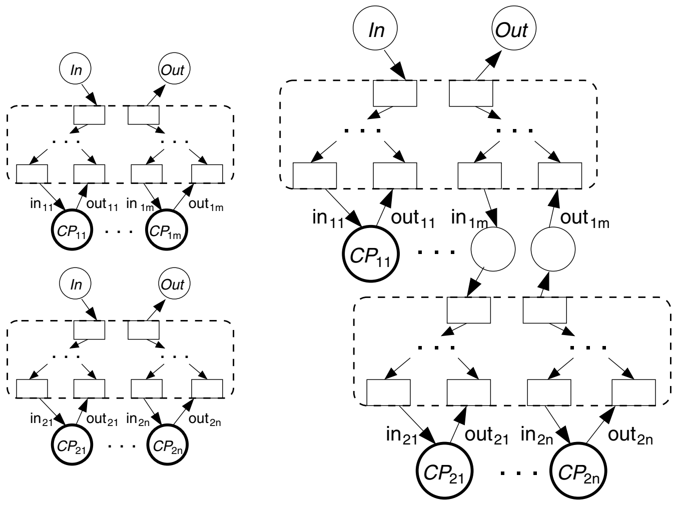


**Fig. 7.** Composing connectors

composed by matching the composition place $CP_{1m}$ in the first CT-net with the input and output places of the second CT-net. The in-arc $in_{1m}$ of $CP_{1m}$ is re-directed to the input place, whilst the out-arc $out_{1m}$ is re-directed from the output place, of the second CT-net. The resulting composite connector has the input and output places of the first CT-net; the composition places $CP_{11}, \ldots, CP_{1(m-1)}$ of the first CT-net and $CP_{21}, \ldots, CP_{2n}$ of the second CT-net. (The input and output places in the second CT-net become dummy places in the composite CT-net.) An example of CT-net composition can be seen in Fig.11, where the *Pipe* and *Cobegin* CT-nets from Fig.6 are composed.

Clearly composition of CT-nets is hierarchical. The resulting composite connector is thus *self-similar* to its sub-connectors, i.e. it has the same structure as each of the latter. This self-similarity is significant because it means that the composition of connectors is strictly hierarchical; it is also algebraic because a composite connector also encapsulates control, just like its sub-connectors. Indeed, composite connectors have the following property:

*Property 2.* Property 1 holds for composite connectors that are composed from primitive operators when no places are shared during composition.

Thus composite connectors can be used in further composition, just like the basic connectors. This is because their control flow is similar to that of the latter and it guarantees that the only control threads returned are the ones that are given as input to the connector.

For self-similarity, the proviso of no shared places during composition must be observed. However, this can be overcome by the use of dummy places that serve as 'memory' places that retain copies of tokens and thus simulate non-sharing of places.

## 5 Behavioural Patterns

We have seen that our connectors encapsulate control, and can be composed into composite connectors. In this section we will show that because they encapsulate control and are generic in terms of arity and parametricity, they can be used to define design patterns [5], more precisely, behavioural patterns. We will show that even a basic connector can be used to define a pattern; whilst a composite connector can be used to define a more complicated pattern.

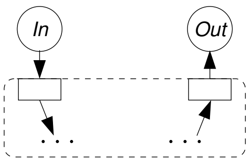


**Fig. 8.** A component

Specifically, a connector can only define the control flow in a pattern; it cannot specify the participants and their roles in the pattern. The participants are of course components, so we need to consider how components are defined and how they are composed by connectors. A component is defined as a net with distinguished *Input* and *Output* places (Fig.8) connected to transitions (boxes) in a Coloured Petri net (dotted box). Such a net is the same as a CT-net (Fig.5) except that it has no composition places. Clearly, like a CT-net, a component net can be composed with a CT-net via the latter's composition places.

To use a CT-net to define a pattern, we need to add constraints that specify the components that participate in the pattern, and their roles, to ensure conformance with the semantics of the pattern. In addition we could also have constraints on the CT-net itself (as we will see later). Thus a pattern is a pair (CT-net, *constraints*), where *constraints* specify the participating components and their roles, and possibly also some restrictions on the CT-net. In other words, just as a (concrete) CT-net is an instance of a template (specified by its arity), a pattern is an instance of a CT-net (specified by its constraints).

### 5.1 Constraints

The Coloured Petri net in a component net (Fig.8) represents the behaviour of the methods[5] in the computation units of the component. Therefore, in a pattern, the constraints on the participating components and their roles are expressed in terms of the names and types of the methods and their parameters in these units. We denote these constraints by $C$.

Constraints on the CT-net in a pattern are constraints on the composition places in the CT-net that are instantiated by the components composed by the CT-net. These can express restrictions on, or adaptations of, the control flow in the CT-net, e.g. adding guards or conditional branching, and can alter the control flow in and out of a composition place. We will denote these constraints by $D$.

Thus a pattern is (CT-net, ($C$, $D$)). $C$ has to specify the roles of the participants as well as the relationships between the participants. We will define $C$ as a pair of pre- and post-conditions $p$ and $q$, i.e. $C = (p, q)$:

[5] These methods have pre- and post-conditions.

(i) The pre-condition $p$ specifies type conditions on the names and parameters of methods in the computation units of the participating components, as well as the relationships between these names and parameters.
(ii) The post-condition $q$ specifies the expected behaviour of the pattern $P$, in terms of the expected behaviour of each participating component.

A pattern $P = (\text{CT-net}, (C, D))$ can only be applied as a composition operator to a set of components if the components collectively satisfy the pre-condition $p$ in $C$. Satisfaction is checked by matching $p$ with the pre-conditions of the methods in the computation units of the participating components. For valid components, the pattern $P$ acts a composition operator with an adaptation by $D$ of the control flow of CT-net.

We have defined a constraint language for patterns. For $C$ constraints, our language has similarities with OCL [26], a constraint language for objects in UML; however, unlike OCL, our language can also be used to define $D$ constraints. For lack of space, we do not give full details of our constraint language, and will only give and explain some of the constraints that we will use.

To implement a pattern $P = (\text{CT-net}, (C, D))$, we need to combine the semantics of CT-nets and the semantics of our constraint language. The $D$ constraints in our constraint language can be implemented in a straightforward manner, since they define control structures which can be easily implemented in a programming language. In contrast, 'implementing' $C$ constraints amounts to automatic specification matching and verification. This requires theorem proving and is undecidable in general. So in our implementation (see Section 6) $C$ constraints are annotations that require manual checking.

## 5.2 Basic Composition Operators

Even a basic connector can be used to define a pattern. Consider the *Sequencer* connector. Its CT-net is shown in Fig.6(a), and Fig.9(a) shows the control flow it encapsulates. By defining suitable constraints we can use the *Sequencer* to define the Chain of Responsibility (CoR) pattern.

According to its description in [5], the intent of CoR is to "avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request; chain the receiving objects and pass the request along the chain until an object handles it". So to define the CoR using *Sequencer*, we need to allow two different control flows, depending on whether control exits after the

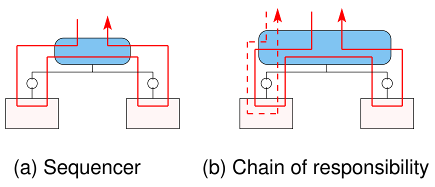


**Fig. 9.** Chain of responsibility

first or the second component successfully handles the request. This is shown in Fig.9(b).

We define CoR = (CT-net for *Sequencer*, $C, D$), where $C$ and $D$ are defined as follows.

For simplicity, we continue to use binary connectors. Therefore, the CoR pattern, applied to two components `C1` and `C2`, requires that the second component `C2` can be used instead of the first one `C1`. For that reason we define the notion of behavioural conformance between the two components [20]. Specifically, for each method in `C1`, `C2` must provide a method with the same i/o parameters, a weaker pre-condition and a stronger post-condition. Thus the pre-condition for CoR, in our constraint language, is:

```
C1.methods->forAll(m1:Method | C2.methods->exists(m2:Method |
                    (m2.input = m1.input and m2.output = m1.output and
                    m1.Pre implies m2.Pre and m2.Post implies m1.Post))
```

The post-condition of the CoR ensures that whenever a method `m` in a component is invoked, the post-condition of `m` is satisfied.

```
C1.methods(invoke).Pre implies C1.methods(invoke).Post or
C2.methods(invoke).Pre implies C2.methods(invoke).Post
```

We need a $D$ constraint that specifies that control reaches the next composition place iff the pre-condition of the component in the current composition place with the given input parameters is satisfied. This is defined as:

```
if(eval(cp(currentIdx).methods(target).Pre,input.value)) then
  return cp(currentIdx).methods(target).output.value
endif
```

## 5.3 Composite Composition Operators

A composite connector can also be used to define a pattern. Consider the composition of the *Pipe* and *Cobegin* connectors (defined in Fig.6). The resulting composite connector is shown in Fig 10. It passes results from $P$ to $S1$ and $S2$. By defining suitable constraints, we can use this composite connector to define the Observer pattern. According to its description in [5], the intent of the

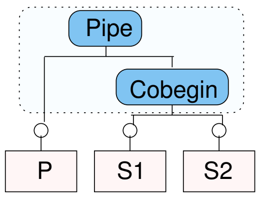


**Fig. 10.** 'Observer'

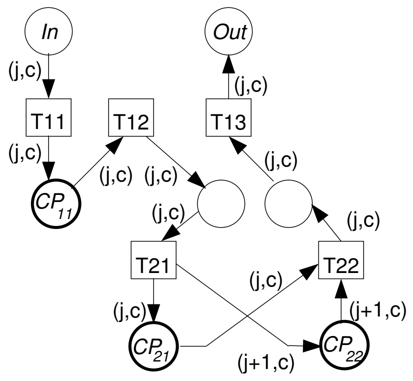


**Fig. 11.** CT-net for Observer

Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically". Therefore the composite connector in Fig.10 acts as an Observer, with a publisher $P$ and two subscribers $S1$ and $S2$.[6]

So we define Observer = (CT-net for *Pipe* + CT-net for *Cobegin*, $C, D$).

The composition of the CT-nets for *Pipe* and *Cobegin* is shown in Fig.11. This is the CT-net for Observer.

The Observer composed from binary *Pipe* and binary *Cobegin* has three composition places; it is therefore ternary.

In addition, $C$ constraint is defined as follows.

The pre-condition of the (ternary) Observer (applied to components `C1`, `C2`, `C3`) requires that some of the methods from the publisher component `C1` can be matched by the methods of both the subscribers `C2` and `C3`. This means that the output of `C1` can be consumed as the input of `C2` and `C3`. Therefore the pre-condition for Observer is:

```
let M, M1: Set(Method)
M1 = C1.methods(all)->select(m:Method |
                             m.Post implies (length(m.output) > 0))
M = M1->select(m1:Method | C2.methods->exists(m2:Method |
                             m2.input includes m1.output) and
                             C3.methods->exists(m3:Method |
                             m3.input includes m1.output))
M->size()>0
```

The post-condition of Observer ensures that the output of the publisher `C1` will actually be used as (part of) the input to both the subscribers `C2` and `C3`. We describe the part of the post-condition that applies to `C2` (a similar one applies to `C3`). This post-condition is defined as:

[6] Of course it would be better if $P$ was an active component.

```
let pos:Integer
pos=C2.methods(invoke).input->indexOf(C1.methods(invoke).output)
C2.methods(invoke).input(pos .. (pos +
    length(C1.methods(invoke).output))).value
    = C1.methods(invoke).output.value
```

The $D$ constraint for Observer is simply empty because the control flow defined by the composite composition connector already satisfies the pattern.

### 5.4 Composing Behavioural Patterns

Defining behavioural patterns as connectors offers the immediate benefit of being able to compose patterns in the same manner that we compose any connectors. However, constraints must be composed correctly. For two patterns $P_1$ = (CT-net1, $C_1$, $D_1$) and $P_2$ = (CT-net2, $C_2$, $D_2$), the resulting composite pattern has $C_1 \wedge C2$ (with renaming) as its $C$ constraints, and has $D_1$ and $D_2$ as its $D$ constraints on CT-net1 and CT-net2 respectively. This kind of compositionality is a result of encapsulation in our model.

For example, we can compose the Observer and CoR patterns into a composite (Fig.12). This composition connector connects a chain of publishers ($P1$ and $P2$) in a CoR to a set of subscribers ($S1$ and $S2$). This composite pattern extends the Observer to multiple publishers. The subscribers are however only interested in the first result, produced by (any of) the publishers. For instance, this pattern may apply to a scenario in which news subscribers wish to receive the first available news bulletin on a particular topic, published by any (of a set of) news agencies they subscribe to.

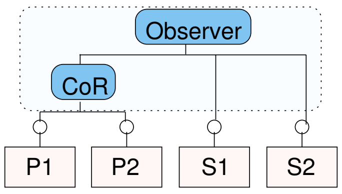


**Fig. 12.** ‘CoR-Observer’

We have defined the composite pattern CoR-Observer with four composition places. The first two, `C1` and `C2`, correspond to the publishers which are constrained by the CoR, whilst the last two, `C3` and `C4`, correspond to the subscribers. Consequently the pre-condition of CoR-Observer must require `C1` and `C2` to act as publishers in a chain of responsibility. We only need to describe the requirements for `C1` w.r.t. the Observer's requirements. This is because in CoR `C2` offers more than `C1` and requires less than `C1`. The pre-condition is:

```
let M, M1: Set(Method)
M1 = C1.methods(all) -> select(m1:Method |
                   m1.Post implies (length(m1.output) > 0))
```

```
M = M1 -> select(m1:Method | C3.methods->exists(m3:Method |
                  m3.input includes m1.output) and
                  C4.methods->exists(m4:Method |
                  m4.input includes m1.output))
M->size()>0   and
C1.methods->forAll(m1:Method |
                 C2.methods->exists(m2:Method | m2.input = m1.input and
                 m2.output = m1.output and
                 m1.Pre implies m2.Pre and m2.Post implies m1.Post))
```

The above conditions specify that the methods of C1 must provide some output and this must be acceptable by the C3, C4 components (a requirement for the Observer pattern). It also means that C2 can be used instead of C1.

The post-condition of CoR-Observer states that when C1 gets invoked its output is consumed by the subscribers, and similarly for C2.

The *D* constraint for CoR-Observer is simply the *D* constraint for the first two composition places:

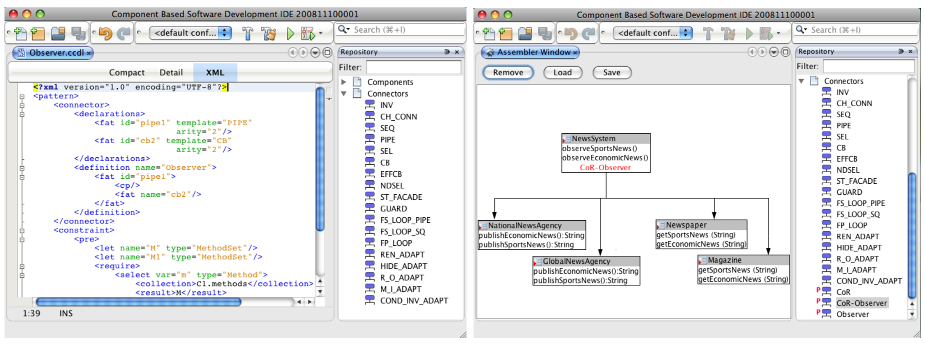
```
let currentIdx : int with 1..2
if(eval(cp(currentIdx).methods(target).Pre,input.value)) then
   return cp(currentIdx).methods(target).output.value
endif
```

## 6 Implementation and Example

The CT-nets we have been using are defined in CPN Tools. Connectors could therefore be defined and composed using CPN Tools. However, CPN Tools cannot handle connectors that are patterns, because of the associated constraints. Therefore, we need to implement connectors ourselves, in a tool that can implement CT-nets as well as constraints for connectors that are patterns. In any case, for connectors (patterns) to be useful, we also need to implement components. So we have started to implement a tool for our component model. The tool supports the idealised component life cycle [19] consisting of: (i) *design* phase, in which components and connectors are designed, implemented and deposited in a repository; (ii) *deployment* phase, in which components and connectors are deployed into a specific application under construction. The tool therefore consists of a *builder* for the design phase, and an *assembler* for the deployment phase.

In the builder, connectors and patterns can be defined and stored in a repository, alongside components. Basic connectors are defined first, and then used to define composite connectors. Design patterns can be defined from basic or composite connectors that have already been defined. By storing all the connectors in the repository, we can reuse them in the deployment phase for many different applications. In this way, as reusable connectors that can be retrieved from a repository, so design patterns become really reusable.

Basic connectors are implemented as Java classes. For composite connectors, we define a connector composition language which is based on XML, which allows us to define the structure of a composite connector in terms of smaller



```xml
<?xml version="1.0" encoding="UTF-8"?>
<pattern>
    <connector>
        <declarations>
            <fat id="pipe1" template="PIPE"
                            arity="2"/>
            <fat id="cb2" template="CB"
                            arity="2"/>
        </declarations>
        <definition name="Observer">
            <fat id="pipe1">
                <cp/>
                <fat name="cb2"/>
            </fat>
        </definition>
    </connector>
    <constraint>
        <pre>
            <let name="M" type="MethodSet"/>
            <let name="M1" type="MethodSet"/>
            <require>
                <select var="m" type="Method">
                    <collection>C1.methods</collection>
                    <result>M</result>
```

(a) Defining the Observer pattern. (b) Building the system using patterns.

**Fig. 13.** Our prototype tool

connectors. We then implement a Java class that takes such structural definitions and realises the desired behaviour through object aggregation and composition of connector objects.

Pattern definition in our repository consists of a connector definition and its associated constraints. For pre-defined (existing) connectors, the connector definition is a unique string for every connector. For (new) composite connectors, the structural definition must be given. The constraints for the connector are then defined and tied to the connector definition.

*D* constraints can affect the control flow of our connectors, and therefore must be executed at runtime. Therefore, in our implementation, *D* constraints are transformed into Java code that is used to generate Java classes to realise the pattern. Indeed, the transformation is possible because *D* constraints are actually control flow structures, e.g. *if-endif* in the *D* constraints of CoR. The Java code in this case is the *try-catch* structure which captures a special exception thrown by the violation of the pre-condition of a component, before invoking the next component. This is because methods in a component throw a special type of exception if their pre-condition is violated. Thus the code of *CoR* pattern class has the code of the *Sequencer* connector class and the *try-catch* code realising the *D* constraint.

Consider the Observer pattern. Fig.13(a) shows its definition using the builder tool. The XML definition consists of two sections that are identified by <connector> and <constraint> tags for composing two basic connectors and defining constraints. It is clear that a binary pipe (*pipe1*) and a binary cobegin (*cb2*) are declared. The connector *pipe1* thus has two composition places, and the connector *cb2* replaces the second composition place, and is thus composed with *pipe1*. The pre-condition of the constraint is also shown in the figure; it is defined and constructed manually.

Once defined, Observer pattern definition is stored into our repository as can be seen in Fig.13(b) (bottom right corner). CoR and composite patterns can be defined in a similar way and stored in the repository.

A pattern, like any connector, can be used to compose components that are valid for the pattern. For example the CoR-Observer composite pattern can be used to build a news system by composing publisher components which are news agencies and subscriber components which are news companies that print news journals. Fig.13(b) shows an example with two news agencies, one national and one global, and two news companies, one printing newspapers and one printing magazines. The news agencies form a CoR and publish news whenever it became available. The news companies simply pick up the news on a topic of interest to them, from the first news agency that can supply that piece of news.

In design phase we build the four atomic components which we then store into the repository. For each atomic component, its XML specification is defined and its computation unit is implemented. We only discuss the `NationalNewsAgency` component, as the other components are similar. Below we present the XML specification of the `NationalNewsAgency` component:

```
<AtomicComponent>
 <Name>NationalNewsAgency</Name>
 <Method>
  <name>publishEconomicNews</name>
  <InParamName> ...
  <InParamType> ...
  <OutParamsName> ...
  <OutParamType> ...
  <Pre>
   <let name="hasMoreEconomicNews" type="Boolean"/>
   <require>hasMoreEconomicNews==true</require>
  </Pre>
  <Post>economicNews != null</Post>
 </Method>
 <!-- Similar for publishSportsNews method-->  ...
</AtomicComponent>
```

The national news agency can publish either economic or sports news. The pre-condition for the economic news states that there is some economic news available. If the pre-condition is satisfied then the result will be non-null. The computation unit of this component is implemented as a Jar file. The implementation indeed realises the component's XML specification. We experimented with JML [12] for annotating our computation units so that designated exceptions are thrown to signify pre- and post-condition violations.

At run-time, as the component is composed with CoR, we need to check that before calling `publishEconomicNews`, its pre-conditions (as specified in the XML description above) are satisfied. As we explained earlier we currently rely on run-time exceptions to check that. If the exception is not thrown, the CoR source code returns with valid output. Otherwise, CoR invokes the next component in the chain. Similar holds for post-conditions. The implementation of the *D* constraints in the source code of the *CoR* is outlined as follows:

```
// References of components e.g. C1 & C2
private List<Component> comps = ...
boolean success = false;
```

```
// Invoke the component in sequence as long as the previous
// invocation fails because of pre-condition is violated
for (i=0; i < comps.size() && success == false; i++) {
  try {
   Object[] res = comps.get(i).invoke(...);
   success = true;
  } catch (PreViolatedException pve) { ... }
...
```

The Java code snippet enables CoR to return control (and data) when an invocation is successful, i.e. `PreViolatedException` exception is not thrown.

The XML specification and the Java implementation for the observer components (the magazine and the newspaper) are defined similarly to the above and due to lack of space we do not present them here.

When all participating components have been defined, before using them into a composition with the CoR-Observer, we (manually) check that the pre-conditions of the CoR-Observer can be satisfied. First, we check that the global news agency conforms to the national one. Because this condition is satisfied, the components form a chain of responsibility. Additionally, the local agency can be used for publishing news to the magazine and newspaper observers because we map `publishEconomicNews` and `publishSportsNews` to `getEconomicNews` and `getSportsNews` respectively. Since the pre-conditions of the CoR-Observer have been satisfied, its post- conditions ensure that the resulting news of the agency that are published first, are transferred successfully to both observers.

Based on the CoR-Observer composition connector and on the atomic components used, a composite component `NewsSystem` is created that will have two methods, `observeSportsNews()` and `observeEconomicNews()` as in Fig.13(b).

## 7 Discussion

The main contribution of this paper is to show how behavioural patterns can be defined and implemented as explicit entities with their own identities that can be deposited in a repository, and reused as often as necessary. This is novel, as far as we are aware, compared to related work in design patterns and software components.

By defining behavioural patterns as composition operators, we have retained the original semantics intended for patterns as defined in [5], i.e. as reusable solutions that can be customised to different circumstances. Our composition operators for patterns are of generic arities, and can be applied to any components that satisfy the constraints. Each application of an operator to a selected set of components represents a customisation of the solution to the context of these components.

Furthermore, our approach (and tool) can be used to define arbitrary behavioural patterns, and not just the ones that are already known in the literature [5]. More precisely, we can define behavioural patterns that involve pre-determined interactions between multiple participating components. Among existing behavioural patterns in [5], besides CoR and Observer, such patterns also include Visitor, State and Strategy. Hence, we can define more composite design patterns such as Strategy-Observer and State-Observer (which extend Observer to multiple publishers but with different publisher selection strategies), and Strategy-Visitor and State-Visitor (which can extend the Visitor pattern with many visitees and visitors), etc.

Behavioural patterns with arbitrary interactions, e.g. Iterator, Mediator and Memento, however, cannot be pre-defined as connectors (and deposited in a repository) *in design phase.* In our component model, such patterns are purely deployment phase artefacts. They have to be defined *ad hoc* from our basic connectors like *Sequencer*, *Selector*, etc., anew for each application. Also in our component model, structural patterns also belong to this category because they can define arbitrary behaviour e.g. Facade and Adapter. Note that, this could involve using stateful versions of our connectors and adapters.

Behavioural patterns that involve only one participant or define no interaction between participants, e.g. Template Method and Interpreter, do not require any composition, and as such they cannot be defined as connectors.

Our approach currently requires manual checking of $C$ constraints. Therefore, patterns need to be used and checked manually. This is a hindrance. However, the effort needed for creating a pattern is a one-off effort, and so it should still pay dividends because of it reusability. Moreover, in future we intend to enhance our $C$ constraints with semantic annotations on component interfaces, and thereby automate constraint checking by implementing a suitable parser and evaluator. We will also study design patterns in a wider scope in order to seek new pattern connectors, thus extending our catalogue of connectors as patterns.

Modelling component-based systems using Petri nets has led to various extensions to Petri nets for different kinds of components. For example, *service nets* [6] are used to describe the behaviour of a web service. Compared to CT-nets, a service net also has input and output places, but not composition places. Composition operators for service nets are not nets themselves, but just rules for obtaining the service net of a composite service from the service nets of the sub-services. In [9] component templates are defined as an extension of Petri nets for describing the behaviour of components. However, no composition operators are defined. Rather, composition occurs via embedding. In [31] Template Coloured Petri nets (TP-nets) are defined for specifying the behaviour of components. Components are processes in a message passing environment such as MPI [22]. However, composition of TP-nets is defined according to a pre-defined script and not according to composition operators.

Coloured Petri nets have been used for modelling patterns of control for workflow systems [28]. This work is very similar to ours in that our CT-nets define patterns of control for component-based systems. However, they do not define composition for patterns. Compositionality of patterns is claimed, but this is actually an *ad hoc* combination, where places and transitions are arbitrarily connected and/or merged.

## 8 Conclusion

In this paper, we have presented an approach to behavioural patterns that we believe can achieve reuse at the levels of patterns as well as code. This is an advance on the state-of-the-art as far as we are aware.

Our implementation is at a preliminary stage, but initial results have provided proof of concept. This has encouraged us to continue to expand our catalogue of connectors and patterns in the builder, with a view to tackling large scale applications in due course. Such applications will allow us to validate our approach, and provide a more convincing case for its practicability and scalability.

We have not investigated connectors for dynamic or run-time composition. Currently our component model defines composition in design and deployment phases, but not dynamic composition at run-time. Composition is static mainly because we insist on defining composition operators. Such operators are harder to define for run-time phase, and so far we have not investigated them.

Another future direction that we would like to pursue is to investigate the definition and use of patterns in specific domains. In this context, we will develop our approach in the European industrial project CESAR [2]. In particular, we will use it to provide design patterns for efficient composition of components into embedded systems in the avionics domain.

## References

1. Bosch, J.: Specifying frameworks and design patterns as architectural fragments. In: TOOLS 1998, p. 268. IEEE Computer Society, Los Alamitos (1998)
2. CESAR project, `http://www.cesarproject.eu/`
3. Denmark CPN Group, University of Aarhus. CPN tools - computer tool for Coloured Petri Nets, `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`
4. DeMichiel, L., Keith, M.: Enterprise JavaBeans 3.0. Sun Microsystems (2006)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Design. Addison-Wesley, Reading (1995)
6. Hamadi, R., Benatallah, B.: A Petri net-based model for web service composition. In: Proc. 14th Australasian Database Conf., pp. 191–200 (2003)
7. Hammouda, I., Koskimies, K.: An approach for structural pattern composition. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 252–265. Springer, Heidelberg (2007)
8. Heineman, G.T., Councill, W.T. (eds.): Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, Reading (2001)
9. Janneck, J.W., Naedele, M.: Modeling hierarchical and recursive structures using parametric Petri nets. In: Proc. Adv. Simulation Tech. Conf., pp. 445–452 (1999)
10. Järvinen, H.-M., et al.: Object-oriented specification of reactive systems. In: Proc. ICSE 1990, pp. 63–71. IEEE Computer Society Press, Los Alamitos (1990)
11. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, 2nd edn., vol. I. Springer, Heidelberg (1996)
12. The Java Modeling Language, `http://www.cs.iastate.edu/~leavens/JML.html`
13. Lau, K.-K., et al.: Composite connectors for composing software components. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 18–33. Springer, Heidelberg (2007)

14. Lau, K.-K., Ntalamagkas, I.: A compositional approach to active and passive components. In: Proc. 34th EUROMICRO SEAA, pp. 76–83. IEEE, Los Alamitos (2008)
15. Lau, K.-K., Ornaghi, M., Wang, Z.: A software component model and its preliminary formalisation. In: Proc. 4th FMCO, pp. 1–21. Springer, Heidelberg (2006)
16. Lau, K.-K., Ornaghi, M.: Control encapsulation: a calculus for exogenous composition. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 121–139. Springer, Heidelberg (2009)
17. Lau, K.-K., Taweel, F.: Data encapsulation in software components. In: Proc. 10th CBSE, pp. 1–16. Springer, Heidelberg (2007)
18. Lau, K.-K., Velasco Elizondo, P., Wang, Z.: Exogenous connectors for software components. In: Proc. 8th CBSE, pp. 90–106. Springer, Heidelberg (2005)
19. Lau, K.-K., Wang, Z.: Software component models. IEEE Trans. Software Engineering 33(10), 709–724 (2007)
20. Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A type theory for software architectures. Tech. Report UCI-ICS-98-14, University of California, Irvine (1998)
21. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE TSE 26(1), 70–93 (2000)
22. Message Passing Interface (MPI) Forum, `http://www.mpi-forum.org/`
23. Meyer, B., Arnout, K.: Componentization: The visitor example. IEEE Computer 39(7), 23–30 (2006)
24. Mikkonen, T.: Formalizing design patterns. In: Proc. ICSE 1998, pp. 115–124. IEEE Computer Society, USA (1998)
25. OMG. UML 2.0 Infrastructure Final Adopted Spec. (2003)
26. OMG. Object Constraint Language, OCL (2006)
27. Riehle, D.: Composite design patterns. In: Proc. OOPSLA 1997, USA, pp. 218–228. ACM, New York (1997)
28. Russell, N., et al.: Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-31 (2006)
29. Sun Microsystems. JavaBeans Specification (1997), `http://java.sun.com/products/javabeans/docs/spec.html`
30. Szyperski, C.: Universe of composition. Software Development (2002)
31. Tsiatsoulis, Z., Cotronis, J.Y.: Testing and debugging message passing programs in synergy with their specifications. Fundamenta Informatica 41(3), 341–366 (2000)
32. Velasco Elizondo, P., Lau, K.-K.: A catalogue of component connectors to support development with reuse. Journal of Systems and Software (2010)
33. Vlissides, J.: Composite design patterns (They Aren't What You Think). C++ report (1998)
34. Wydaeghe, B., Vanderperren, W.: Visual component composition using composition patterns. In: Proc. TOOLS 2001, pp. 120–129. IEEE Computer Society, Los Alamitos (2001)
35. Yacoub, S.M., Ammar, H.H.: UML support for designing software systems as a composition of design patterns. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, p. 149. Springer, Heidelberg (2001)