Using Petri Nets for Modeling Enterprise Integration Patterns

Dirk Fahland[1] and Christian Gierds[2]

[1] Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
d.fahland@tue.nl

[2] Humboldt-Universität zu Berlin, Department of Computer Science, Unter den Linden 6, 10099 Berlin, Germany
gierds@informatik.hu-berlin.de

Abstract. *Enterprise Integration Patterns* are a collection of widely used patterns for integrating enterprise applications and business processes. These patterns informally represent typical design decisions for connecting enterprise applications.
For the set of patterns collected by Hohpe and Woolf in “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions” we presented for each pattern the conceptual translation into a *Coloured Petri Net* (CPN). We then show, how to apply these CPN realizations for defining a formal model based on a system specification using Enterprise Integration Patterns, which allows us to exploit the full power of analysis techniques and range of application for CPN.

Keywords: integration, middleware, Enterprise Integration Patterns, Coloured Petri nets

1 Introduction

In the last decades, companies have developed a multitude of software applications. The interest of reusing them is not only a matter of convenience, but also of costs. Reuse of applications often occurs in connection with third parties—another department, or even another company.

In order to allow applications with different backgrounds—viz. different data types and interfaces—to communicate, all parties have to integrate a communication solution.

In their best practices book *Enterprise Integration Patterns* [1], Hohpe and Woolf have collected a widely used and accepted collection of integration patterns allowing for easier implementation of a communication infrastructure. They identify four possible solutions for solving communication: file transfer, shared database, remote procedure invocation, and messaging.

The patterns presented in the book are typical concepts used when implementing a *messaging system* and have proved to be useful in implementation. They can cope with the asynchronous nature of message exchange and the facts,

that “Networks are unreliable”, “Networks are slow”, “Any two applications are different”, and “Change is inevitable.” On the other hand, the modular nature of patterns allows them to be used efficiently in new implementations.

In the following we present all patterns described by Hohpe and Woolf and give a realization as Coloured Petri Net. Thus we have Petri net building blocks allowing us to model a messaging system as Petri net based on a design given by Enterprise Integration Patterns.

Our goal is to provide means for analysis of a messaging system. Our Coloured Petri Net realization provides a blue print for implementation that already allows us to check properties of the messaging system.

We have to accept one limitation though: Messaging systems are able to cope with highly dynamical situations. Especially, the handling of application instances created at run-time is normally not an issue for messaging systems. As we use Petri nets for modeling we are restricted to fixed topologies. A dynamically changing infrastructure can only be modeled to some extent. However, this issue applies only to very few patterns.

Related Techniques The patterns described by Hohpe and Woolf are not building blocks of a modeling language, but they describe typical *concepts* in designing a messaging system; thus they are an *informal specification language*.

There exist elaborated modeling techniques like the Business Process Model and Notation [2] (BPMN) or the Workflow Patterns defined by van der Aalst et al. [3]. Enterprise Integration Patterns complement these notions by a set of typical designs found in a messaging infrastructure. Since BPMN and Workflow Patterns are closely related to Petri nets, we decided to use Coloured Petri Nets to model EIP, which allows transferring the results to these modeling paradigms.

2 Enterprise Integration Patterns

Before presenting the patterns we shortly introduce Coloured Petri Nets (CPN) [4]. Coloured Petri Nets are used to model distributed, data-depending systems. A Coloured Petri Net has passive elements called places holding data and active elements called transitions processing data. Each piece of data, called token, is typed by a color and represented by a variable-free expression. A place is typed as well and it can only hold token of the same color. Arc inscriptions describe, which kind of token is consumed or produced by a transition. A guard may restrict the execution (firing) of a transition.

Before giving a formal definition of a Coloured Petri Net, we have to introduce *multisets*. In contrast to a regular set, a multiset can contain an element more than once. Formally, for a set C, the multiset M is a function $M : C \to \mathbb{N}$ assigning each element of C its number of occurrences in M. We use C_{MS} to indicate the set of all multisets over a set C.

Let $C = \{a, b, c, d\}$, then the multiset containing 2 times b, 1 time c, and 2 times d can be written as $[b, b, c, d, d]$, or alternatively as $2`b + 1`c + 2`d$.

Definition 1 (Coloured Petri Net).
A Coloured Petri Net *is a tuple* $CPN = (\Sigma, P, T, F, N, C, G, E, I)$ *where*

- Σ *is a finite set of types (*color sets*),*
- P *and* T *are finite and disjoint sets of* places *and* transitions*,*
- $F \subseteq P \times T \cup T \times P$ *is the* flow relation *between places and transitions,*
- C *is a* color function $C : P \to \Sigma$ *assigning each place a color and thus a type,*
- G *is a* guard function *from* T *into expressions, such that* $\forall t \in T : \mathit{Type}(G(t)) = \mathit{Boolean} \wedge \mathit{Type}(\mathit{Var}(G(t))) \subseteq \Sigma$ *(viz. a guard evaluates to true or false and the types of the used variables* $\mathit{Var}(G(t))$ *are included in the given color set* Σ*),*
- E *is an arc expression function from* F *into expressions, such that* $\forall a \in F : \mathit{Type}(E(a)) = C(p(a))_{MS} \wedge \mathit{Type}(\mathit{Var}(E(a))) \subseteq \Sigma$ *(viz. an arc expression must evaluate to the same type/color as the adjacent place* p *(*$p(a)$ *means the place of arc* a*) and the types of the used variables are included in the given color set* Σ*), and*
- I *is an initialization function from* P *into variable-free expressions, such that* $\forall p \in P : \mathit{Type}(I(p)) = C(p)_{MS}$ *(viz. each token/expression on* p *is a multiset over the color of* p*).*

The state of a CPN is defined by a marking. A *marking* is a function M from P into variable-free expressions, such that $\forall p \in P : \mathit{Type}(M(p)) = C(p)_{MS}$. Thus a marking assigns to every place a possibly empty multiset of expressions (and thus values). Each expression is called a *token* on the corresponding place and it must have the type color as the place. The initialization function I in Def. 1 is the initial marking of a CPN.

The behavior of a CPN is defined by firing transitions. In order to fire an transition t we first need a *binding* of all variables related to t (viz. all variables in labels of arcs connect with t, and the variables of the guard of t). A *binding* of t assigns to every variable of t a type-correct color, and the guard of t must evaluate to true. An arc expression $E(p,t)$ for a transition t and one of its preplaces p evaluates under binding b to $E(p,t)\langle b\rangle$. A transition t is *enabled* to fire for some binding b, if for all preplaces p of t (viz. $(p,t) \in F$) the number of tokens required by the corresponding arc label is present on the preplace under binding b: $E(p,t)\langle b\rangle \leq M(p)$. When a transition t fires with binding b in a marking M_1, then we reach a new marking M_2:

$$\forall p \in P : M_2(p) = M_1(p) - E(p,t)\langle b\rangle + E(t,p)\langle b\rangle \,.$$

Thus the tokens of the preplaces are removed, and new tokes are produced on the postplaces.

For defining colorsets and variables, we use the following notation:

- `colset <name> = <type>;` defines a new color set of type `<type>` with name `<name>`.

– `var <name>: <colorset>;` defines a new variable of the color set `<colorset>` with name `<name>`.

For the patterns we ultimately use the notation used by *CPN Tools* [5]. An example is depicted in Fig. 1. As usual, places are depicted as circles, transitions as rectangles, the flow relations with arcs. If the color of a place is of interest, we write it close to the place in italics (*Fifo* for the place p2p chan). Each arc is labeled with an expression that can contain typed variables like l and x. More complex labels might even express function application like ins l x, meaning $ins(l, x)$. Necessary colorset and variable definitions are placed in a separate box. Each pattern consists of two parts: the *contents* of the pattern describing a particular functionality, and the *context* in which the pattern may be applied. The contents is highlighted by a gray rectangle.

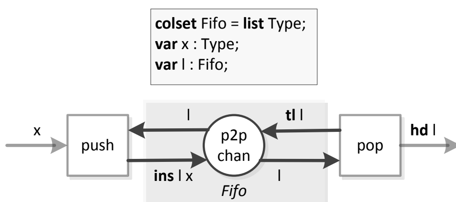


Fig. 1: Example of CPN pattern (Point-to-Point Channel)

In the example, the left transition push binds an expression to variable x of type Type and a corresponding list being of type Fifo to variable l. This type of l is actually a list of elements of Type as we can see by the colset declaration of Fifo. A list provides some functions for manipulation of a list, like ins l x for inserting element x into list l; tl l for return the tail of a list l; and hd l for return the head element of a list l.

When transition push fires, it consumes the value x (from some place) and adds it as last element to the list of values on place p2p chan. Similarly, firing transition pop removes the first element x from the list of values on p2p chan (and puts it on some place) and the remaining list of values back on p2p chan.

Next we present the patterns presented by Hohpe and Woolf. For each we provide a short description and how we can realized it as CPN. We follow the books structure [1, p. xlvii] and refer to it for more details especially on the description and program implementation of these patterns.

2.1 Pattern overview

We now present the actual patterns. We give a short description of each pattern, and describe its realization as a Petri Net. Partially we have to refer to other patterns for realization, as some patterns describe merely concepts and are not fixed in their realization.

We start by describing *message channels*, the fundamental infrastructure of a messaging system. A simple form of a message channel can connect two applications directly, but also broadcasts, multicasts and bus-like communication can be realized with message channels. A single *messages* transports a piece of data, but a specialized form may also carry commands for execution or an event for logging. *Pipes and filters* as well as *message routers* are meant for influencing message content and direction. A filter can drop unwanted messages, a message router can direct message based on a message's content or a system's state to a particular destination, and a pipe as a special form of message channel connects these components. A *message translators* converts messages like transforming from one data format into another or extracting only necessary parts of a message. Finally, a *message endpoint* connects an application to a messaging system. The number after each pattern name refers to the page in *Enterprise Integration Patterns* [1], where the corresponding pattern is described in more detail.

Message Channel A message channel connects applications and allows them to transmit data by connecting a sender with a receiver. This normally follows a first-in-first-out semantics; however, messaging systems may work differently.

Pattern	*Message Channel (60)*
Pictogram (HW)	
Description	A point-to-point or publish-subscribe connection for applications using messaging. A channel acts as logical address, thus the actual receiver is determined by the messaging system.
Realization	The concrete CPN model of a channel depends on additional channel properties. For asynchronous out-of-order communication we simply use a (bounded) place for communication. Otherwise we have to implement a queue or what ever behavior is required (cf. below).

Messaging Channels Messaging channels can occur in different shapes, depending how applications should be connected and what qualities a channel should provide.

Pattern	*Point-to-Point Channel (103)*
Pictogram (HW)	

Description

A unidirectional channel (without additional properties is technically a bucket. The sender puts data on the channel, the receiver takes it from the channel. If multiple receivers are connected to the channel, the actual recipient is not necessarily determined. However, a particular message is taken only by one receiver.
The order of messages is a matter of implementation. In existing messaging systems we can normally assume the channel to be a queue, thus the messages are received in order of their sending.

Realization

A *Point-to-Point Channel* is a special pattern for a *Message Channel (60)*, thus in a Petri Net a Point-to-Point Channel is basically a place, for which a sender puts token on this place, and a receiver removes token from this place. Depending on further requirements, we have to organize the correct handling for this place, as shown here.

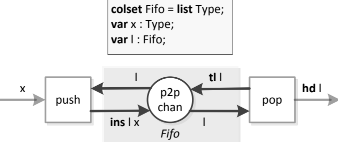


Fig. 2: Point-to-Point Channel with fifo semantics

The above figure shows an implementation of a *fifo channel* as CPN. The actual channel is organized as a place containing a single list. We can send a new message to the channel (left transition) by taking the momentary list l and a new message x, and inserting x at the end of l via function ins l x. We can receive a message from the channel (right transition) by taking the momentary list l, putting back the tail of l via function tl l and keeping the head element of l via function hd l. Normally, the channel should be initialized with an empty list 1`[].

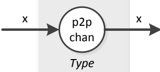


Fig. 3: Point-to-Point Channel with out-of-order semantics

A structurally simpler implementation can be achieved, when the order of messages is not important. Then it is sufficient, if we put a message x on the channel place for sending, and remove any message x from the channel place for receiving x.

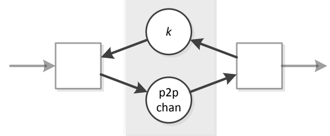
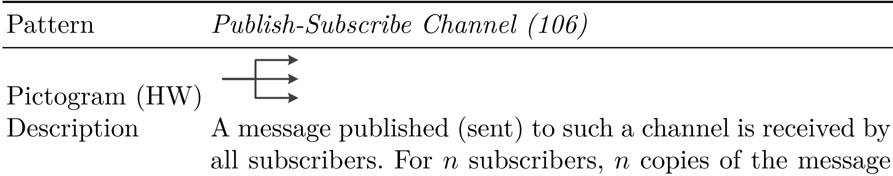


Fig. 4: Abstracted Point-to-Point Channel

For the purpose of model checking it might be interesting to abstract from data. Thus, we have only one color or a classical place/transition net with black tokens. In such a net, the order of messages is unimportant, as single pieces of data cannot be distinguished. However, we still have to model capacity of a buffer. An example is shown above, where we have a buffer with capacity k. We use a complementary place to the actual channel place with initially k tokens. When a message is put on the channel, a token from the capacity place is removed. Removing a message from the channel put a token back on the capacity place. If we put k messages on the channel without removing them, then the channel is full as the complement place is empty. We then cannot send any further message over this channel until at least one message was received and a token was placed back on the complementary capacity place.



Pattern	*Publish-Subscribe Channel (106)*
Pictogram (HW)	
Description	A message published (sent) to such a channel is received by all subscribers. For n subscribers, n copies of the message have to be provided.

Realization

In contrast to the Point-to-Point Channel above, every receiver, which has subscribed to a channel, must receive every message. Thus a message published on the channel must be copied for every recipient.

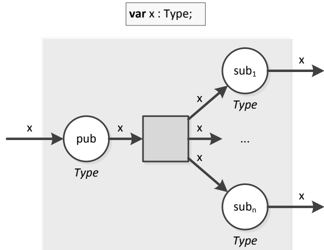


Fig. 5: Publish-Subscribe Channel

Pattern	*Datatype Channel (111)*

Pictogram (HW)

Description

In order to process incoming messages correctly, a message should be typed. Thus a data type channel transports only messages of a certain type. In contrast, a *Point-to-Point Channel* may transport messages of arbitrary type.

Realization

A Datatype Channel is a typed Point-to-Point Channel. Thus their CPN realizations are similar.

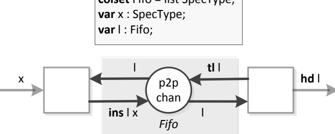


Fig. 6: Datatype Channel

The main difference is that the place p2p chan of a Datatype Channel has a particular pre-specified type where in an untyped channel, the place has not any type. Note however that in a CPN each place has a type. Thus, if needed, untyped channels could be described as having a general type being the union of all specific types.

Pattern	*Invalid Message Channel (115)*
Pictogram (HW)	
Description	Because of missing headers or wrongly formatted data a message might be invalid. Instead of discarding or ignoring an invalid message, such a message may be forwarded to a dedicated channel allowing later handling or logging of that message.
Realization	This is just a special type of channel. Its CPN realization is identical to a Point-to-Point Channel.

Pattern	*Dead Letter Channel (119)*
Pictogram (HW)	
Description	When a message cannot be delivered, e. g. when the receiver is no longer available, or the channel was closed, the message can be rerouted to a Dead Letter Channel. This was, the messaging system does not get cluttered up by left-over messages.
Realization	This is a special type of channel. The messaging system decides, how and when to move a message to a Dead Letter Channel. Its CPN realization is identical to a Point-to-Point Channel.

Pattern	*Guaranteed Delivery (122)*
Pictogram (HW)	
Description	For asynchronous message exchange messaging system usually use a *store-and-forward* approach allowing to buffer messages. This normally happens in volatile memory that is prone for the loss of power or other failures. Guaranteed delivery is an implementation approach that allows to store messages persistently in the messaging system, thus allowing a guaranteed delivery even in case of a problem of the messaging system.

Realization — Our model uses Guaranteed Delivery as default. We would have to model error cases explicitly, if we would need it for analysis. A possible implementation is shown below in Fig. 7.

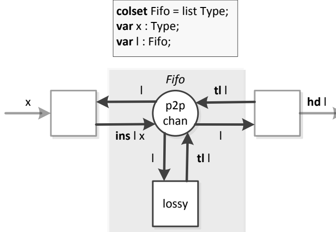


Fig. 7: Lossy Point-to-Point Channel

In case we want to abstract from data, a lossy channel has to remove arbitrary tokens from the channel place. In case we assume a capacity for the channel (cf. Point-to-Point Channel), we remove a token from channel and increase the capacity by putting a token on the complementary place, as shown in Fig. 8.

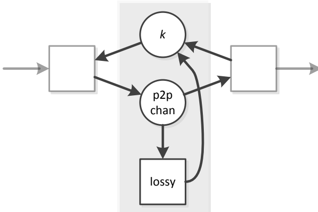
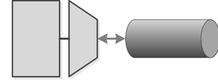


Fig. 8: Abstracted lossy Point-to-Point Channel

.

Pattern	*Channel Adapter (127)*
Pictogram (HW)	
Description	A Channel Adapter connects an application to a messaging channel. Instead of changing the application, the adapter is able to access the application's API or data and thus transparently connects the application to a messaging system.

Realization

The realization depends on the actual functionality of a Channel Adapter. In its simplest form it simply wraps some data into a message and puts it on a channel, or it provides data from a message to the application.
As an effect, we can convert synchronous calls to asynchronous ones and vice-versa.

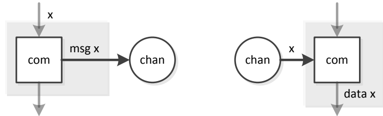


Fig. 9: Outbound and inbound Channel Adapter

Pattern — *Messaging Bridge (133)*

Pictogram (HW)

Description

A Messaging Bridge shall connect two different messaging systems. Even when using the same underlying technology, message formats of the two messaging system might need to be translated into each other.

Realization

Since each messaging system has a preferred message format, a messaging bridge must convert a message of system 1 into a message of system 2 and vice-versa.

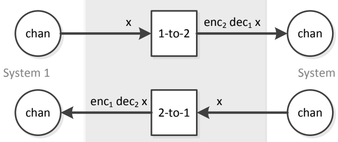
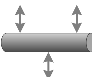
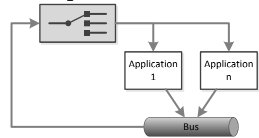


Fig. 10: Messaging Bridge between two systems

The figure shows an example for two channels both of system 1 on the left and system 2 on the right. The messaging bridge provides functions for decoding and encoding messages for both systems. Thus the bridge decodes a message from one system into an intermediate format and encodes it then for the other system. Thus we can also use channel adapters or messaging endpoints to implement such behavior.

Pattern — *Message Bus (137)*

Pictogram (HW)	
Description	A Message Bus acts as central actor for different applications. Each application has only to know about the Message Bus that forwards messages and commands correctly to an appropriate party. Thus the Message Bus provides a combination of “a canonical data model, a common command set, and a messaging infrastructure”.
Realization	The actual realization of a Message Bus depends on patterns used for implementation.  Fig. 11: Message Bus realization using patterns One idea is to provide a central *bus* channel for every application. Thus this is the only channel an application has to be aware of. Based on the message content a message router (cf. pattern *Message Router (78)*) decides where to send the messages.

Message A message is the atomic unit transmitted by a message channel. Data may have to be chunked into smaller packets. Each packet is sent as one message. A sender has to cut data into small pieces, in order to send this data. Accordingly, a receiver has to put together the small pieces, in order to get the data. In this subsection we focus on the different purposes of a message—carrying a command, a document, or an event— as well as certain patterns implying importance to certain messages.

Pattern	*Message (66)*
Pictogram (HW)	
Description	A message is an atomic piece of information sent over a message channel. Nevertheless a message might be structured. Typically a message contains at least header and body that can be structured further. We can partition data into messages in order to be sent.

Realization	Given the means of Coloured tokens, structured message types can be translated into certain colors. Depending on the aim of modeling, a message can be abstracted to a black token for the purpose of analysis.

Message Construction The following patterns are about *message intent*, *returning a response*, *huge amounts of data*, and *slow messages*.

Pattern	*Command Message (145)*
Pictogram (HW)	C
Description	Running a command locally is easier to handle than a remote procedure call. Thus a command might be encapsulated in a message, forwarded to the called application and there run locally.
Realization	A Command Message is merely a concept and highly depends on the actually setting and the techniques used. The following example defines an enumeration of all valid commands and arguments are provided as a list of strings.

```
colset Commands = with Cmd1 | ... | CmdN;
colset Arguments = list string;
colset CommandMessage =
       record cmd:Commands * arg:Arguments;
```

Pattern	*Document Message (147)*
Pictogram (HW)	D
Description	A Document Message encapsulates data into a message, such that it can be transmitted from one application to another.
Realization	A Document Message is merely a concept and highly depends on the actually setting and the techniques used. Similar to a command message we can define structured colorsets or a string colorset to represent a document.

Pattern	*Event Message (151)*
Pictogram (HW)	
Description	An Event Message realizes notification of other applications. Here, timing is more crucial than for other message types. A receiver of such a message is most likely some kind of observer, which logs or reacts to the event.
Realization	An Event Message is merely a concept and highly depends on the actually setting and the techniques used. Such a message may contain a string with a log message, or similar to a command message an enumeration type for a set of predefined events and additional arguments.

Pattern	*Request-Reply (154)*
Pictogram (HW)	
Description	This is a communication pattern allowing two-way communication between two applications. Especially one application makes an request and the second application replies on a separate channel. Invocation of the requester might be a synchronous block, or using asynchronous callback.
Realization	As Request-Reply is a communication pattern, realization depends on the actual goal of the request-reply interaction, thus messages exchanged and channels used.

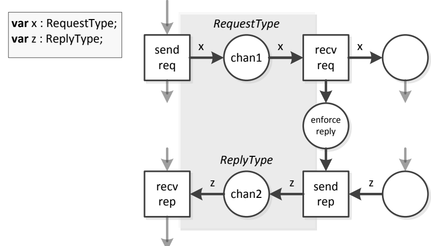


Fig. 12: Request-Reply communication

One possibility for realizing this pattern is the use of two Point-to-Point Channels. On the left-hand side of the figure an application wants to send a request and waits for the reply. The application on the right-hand shall process the request and provide an appropriate reply. We use two distinguished channels: chan1 for sending the request, chan2 for sending the reply. In order to bundle these two channels, we add a control-flow place enforce reply. After a request, a reply must return.
The dependency between request message x and reply z may be arbitrary loose. However it is likely that a *Correlation Identifier (163)* correlates both messages.
If we want to be more flexible, such that even applications unknown to the processing party can send a request, the requesting application may add a return address (see below) in the request. Thus the reply does not go statically back to the requesting party, but the reply is routed dynamically where needed.

Pattern	*Return Address (159)*
Pictogram (HW)	
Description	When an application may receive requests from multiple parties, then it has to send the reply to the appropriate sender. A requester can add a Return Address to the messages header, such that the application knows, where to send the reply to.
Realization	This pattern relates to reference passing. When assuming a static setting, the return address can be either stored separately or kept in the original request. When sending the answer, we have guarded choices, where a guard simply checks, if the return address fits to a certain channel.

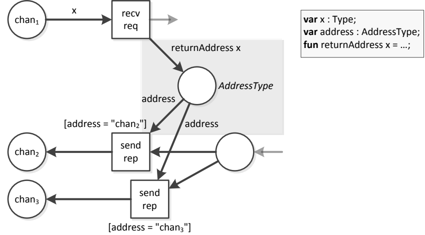


Fig. 13: Using Return Address for selecting recipient

In a dynamic setting, where not all applications are known at design time, the reply may be sent to a message bus. In order to arrive at the right destination, the reply needs to include the return address as recipient.

Pattern	*Correlation Identifier (163)*
Pictogram (HW)	
Description	A Correlation Identifier is used to correlate sent and received messages, when dealing with a multitude of message. For instance, a correlation identifier may by used to correlate a reply to a corresponding request. This is a message header information.
Realization	As a Correlation Identifier is part of a message, realization is purely on the data level. Guards can be used to select correlated messages.

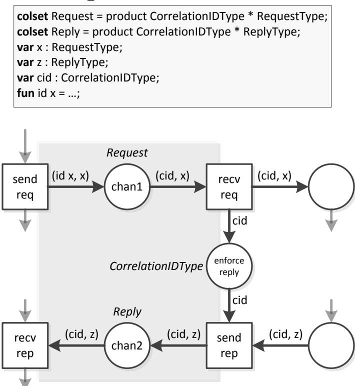
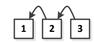


Fig. 14: Using Correlation Identifier in Request-Reply

In Fig. 14 we apply the idea of message correlation on the Request-Reply pattern. When sending request x, function id creates a correlation identifier for the request. Subsequently, only tuples with the correlation identifier cid as first element are used—in the request as well as in the reply. Additionally, the enforce reply place keeps a copy of all identifiers, such that only a reply with a valid correlation identifier can be sent back.

Pattern	*Message Sequence (170)*
Pictogram (HW)	
Description	When data is too much to be sent in one message, data has to be split into several messages. A solution is to use a sequence identifier for messages belonging together, a position identifier for the order of messages, and size or end indicator.
Realization	This pattern is also highly related to the actual data of messages. However, we have to split and join single messages in order to handle large data.

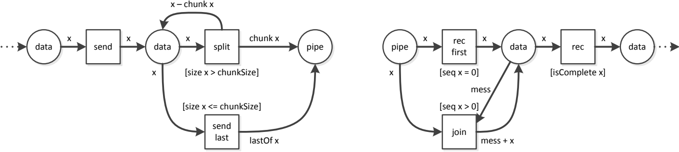


Fig. 15: Sending and receiving a Message Sequence

On the left data is split. Each single message is also enriched with the necessary sequence number and position. On the right incoming messages are joined again into a single data item.
A corresponding definition for the data transported might look as follows:

```
colset seqNo = INT;
colset chunkPos = INT;
colset chunk =
        record seq:seqNo * pos:chunkPos *
               data:Type;
```

For each chunk we provide a sequence number. All chunks with the same sequence number represent part of the same data. The position is literally the position in the sequence of chunks. For the actual data chunk we provide a corresponding field as well.
There are several possibilities, how the receiver can recognize, whether the last chunk has passed. Either the chunks arrive in order, then a special last chunk message can be sent, or we have to store the number of all chunks, either separately or as additional field in each chunk.

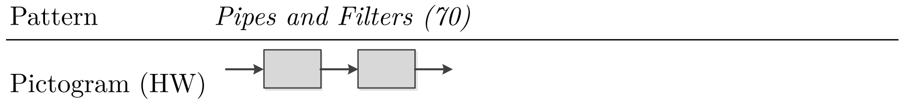
Pattern	*Message Expiration (176)*
Pictogram (HW)	

Description	A message's content or its validity might expire after a certain amount of time. Then the message is either ignored or rerouted to a Dead Letter Channel.
Realization	We could introduce timing constraints to Petri Nets. Otherwise we can non-deterministically decide to invalidate a message with Message Expiration.

Pattern	*Format Indicator (180)*
Pictogram (HW)	
Description	As an application evolves, new data formats may be introduced. To make messaging more robust to changes in data format, a Format Indicator should be included in messages. This way existing channels can be reused and a receiver knows, how to distinguish old and new data formats.
Realization	The Format Indicator pattern deals only with format of data, which we can exploit in subsequent patterns translating different data formats. However, we focus on flow of control and data, and the semantic aspects of data are orthogonal to our focus.

Pipes and Filters, Message Router Pipes and Filters are a form of indirection between sender and receiver. They allow to process, validate, or transform messages. A combination of several pipes and filters is also possible.

Routing applications abstract from the need to select a message channel and a receiver in advance. An application may simply send a message to a message router that then decides (on-the-fly) where the message should be sent to.

Pattern	*Pipes and Filters (70)*
Pictogram (HW)	

Description — A filter (gray box) is a simple processing function with one inbound and one outbound pipe (arc). A pipe simply connects the output of one filter with the input of a second filter. Thus complex processing of messages can be modeled by connecting a series of filters. (N.B. This restriction can of course be relaxed.) Filters might be implemented on different machines. A pipe is a special form of message channel.

Realization — In our setting, a filter is a transition processing a token from a preplace and outputting the result on a postplaces. Accordingly, these places then act as pipes. Depending on the transformation, it might be realized by a high- level function arc-inscription, or the transition might be refined to fulfill the transformation.

In Fig. 16 we have two filters: the first one filters a message `x` resulting in `f x`; the second one results analogously in `g y`.

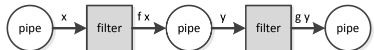


Fig. 16: Pipes and Filters

Issue: A pipeline architecture allows processing of messages concurrently. The first filter immediately starts processing the second message, when it has completed the first one. It does not have to wait for subsequent filters to finish. If we consider a pipe simply as a place, we cannot maintain order of messages without further infrastructure. By default we would only support an out-of-order processing of messages or we have to define a FIFO pipe like in the *Point-to-Point Channel* pattern above.

Pattern — *Message Router (78)*

Pictogram (HW)

Description — Extension of the Pipes and Filters idea. Filters are not simply connected by pipes, but the choice for a filter is made by a Message Router depending on certain conditions. A message router therefore can be seen as a special kind of filter.

Realization — For a given input pipe and one of several output pipes we have to connect the input with one output. The pattern can be realized by firing a transition with a corresponding condition. All conditions together should be mutually exclusive for deterministic behavior, and the should cover all possibilities, such that no message can get stuck in the message router. The conditions can be used as guards for the transitions forwarding a message to a certain pipe.

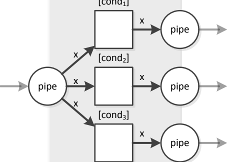


Fig. 17: Message Router

Please note that we are a bit sloppy about the conditions in Fig. 17. The condition does not need to depend on the message x, but the “condition” may be a round-robin or a time-dependent choice of a destination pipe. Thus, the condition may be extended or replayed by control places.

Message Routing The following routing patterns are specializations of the Message Router pattern. They differ in how the destination pipe is chosen. Furthermore, we introduce more complex patterns describing important principles often expressible by a combination of basic patterns.

Pattern	*Content-Based Router (230)*
Pictogram (HW)	
Description	Depending on a message’s content, the message might be routed to different destinations.
Realization	In the realization we have different transitions each checking for a certain condition depending on a message’s content. If a condition is fulfilled a message is put into the appropriate pipe.

Fig. 18: Content-Based Router

Pattern	*Message Filter (237)*
Pictogram (HW)	
Description	A Message Filter checks messages on a channel for a certain criterion. If a message fulfills the criterion, the message is forwarded, otherwise dropped.
Realization	The pattern can be realized by two alternative transitions: one that forwards a valid message, another that drops an invalid message.

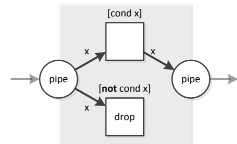


Fig. 19: Message Filter for dropping messages

Pattern	*Dynamic Router (243)*
Pictogram (HW)	
Description	A Dynamic Router acts as central message router for a changing messaging system (viz. actual participating applications and channels). The dynamic router has two aspects: first, it is able to react to newly added or removed channels and address them appropriately; second, even for a fixed set of channels, it may change the conditions, where to send a message based on a rule base.
Realization	For the dynamic router we can only realize the second aspect, the consideration of a rule base for addressing recipients. We are limited to fixed topologies; i. e., where all possible endpoints and channels are known.

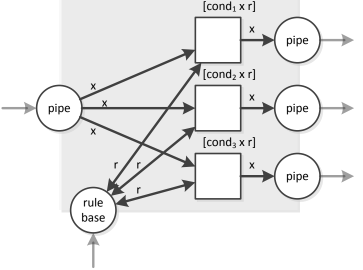
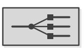


Fig. 20: Dynamic Router using additional rule base

Pattern	*Recipient List (249)*
Pictogram (HW)	
Description	This pattern allows to send a message to multiple recipients based on a dynamically changing list.
Realization	In the realization we assume the set of potential recipients to be fixed. However, depending on a list we decide, whether a recipient receives the message or not.

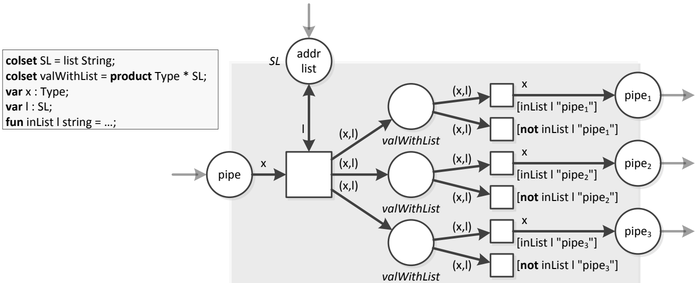
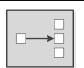


Fig. 21: Recipient List using address list

We first retrieve the address list and make a copy of the message to be sent including the list of addresses for every potential recipient. To facilitate things, let us assume the address list contains the corresponding pipes as recipients. Then for each copy i of the message we have to check, if the corresponding pipe is in the list of addresses ([inList l "pipe$_i$"], meaning forwarding the message to pipe i, or if not, meaning discarding the copy.
Alternatively, the recipient list can be part of the message, and it has to be extracted for deciding, whether a recipient is valid or not.

Pattern	*Splitter (259)*
Pictogram (HW)	

Description — A Splitter breaks a complex message into smaller parts. It does this either iteratively (e. g. by splitting a tree structure into subtrees), or statically (viz. into a fixed number of parts).

Breaking a complex message into smaller parts normally is necessary, when the parts should be processed separately. One of the above router pattern then can direct the parts to the appropriate processing applications.

Realization — The iterative splitter takes a piece of data and breaks it conditionally into smaller parts.

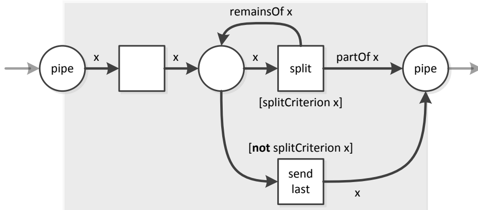


Fig. 22: Iterative Splitter for messages

The static splitter just breaks a piece of data into smaller parts.

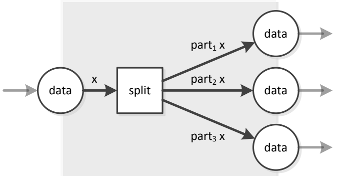


Fig. 23: Static Splitter for messages

Pattern	*Aggregator (268)*
Pictogram (HW)	
Description	An aggregator combines single, but related messages to a complex one in order to allow better processing. Alternatively a result message is updated by incoming message (e. g. for getting the highest return value and discarding all other values).

Realization

In order to realize this pattern, a first message has to be taken as basis for the result and each further message triggers an update of the result, either by cumulation of all data or updating a certain value in the result.

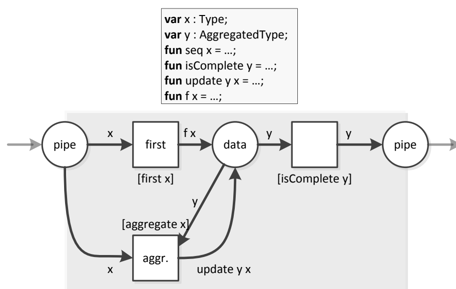


Fig. 24: Dynamic Aggregator

The first arriving `x` is translated into the aggregated type and every subsequent message is processed by the `update` function, until aggregation is complete.
We can also consider static aggregation, where always a fixed number of messages is aggregated.

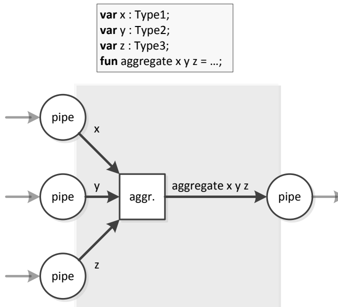


Fig. 25: Static Aggregator for messages

Depending on the aggregation strategy, aggregation can be changed (cf. [1, page 272]).

Pattern	*Resequencer (283)*

Pictogram (HW)

Description

Due to the asynchronous nature of messaging, messages might arrive out of order. A Resequencer then is used to bring messages back into the right order.

Realization

In contrast to an aggregator a Resequencer does not wait for all messages of a sequence, but a Resequencer directly forwards messages being in order. The realization depends mainly on the buffer size and the question, if a buffer overrun should be avoided and how. Otherwise sequence numbers are needed in order to decide, whether messages are in order.
Let us assume, that data is divided into message chunks.

```
colset seqNo = INT;
colset chunkPos = INT;
colset chunk =
        record seq:seqNo * pos:chunkPos *
                data:Type;
```

For each sequence number we must keep track, which was the last message forwarded in order. We assume a place holding this information. A message is only forwarded, if according to this place it is the next message.

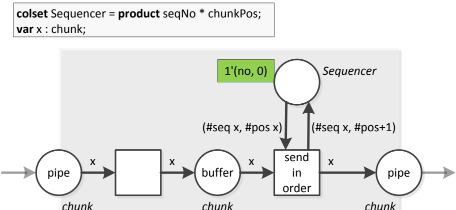


Fig. 26: Resequencer

The Petri Net pattern receives the single chunks and stores them on a buffer place. Additionally we have the tracking place keeping tuples of sequence number and chunk position.
Each sequence of corresponding messages shares the same sequence number no. Initially we assume for each sequence number no a token $1`(no, 0)$ to be available describing that from a sequence of number no, initially only the package with number 0 will be forwarded while other messages are held back in the buffer.
The transition send in order is only able to fire, if one of the chunks x has a corresponding sequence number #seq x, and it is the next wanted chunk #pos x. The chunk is then forwarded and the tracking place holds a token with an increased sequence number.
When we abstract from data for analysis, then this pattern can forward token in any order.

Pattern	*Composed Message Processor (294)*
Pictogram (HW)	
Description	A Composed Message Processor splits a complex message into its components, such that each component can be processed independently (and differently), and puts the result back into a complex message.
Realization	This pattern can be realized be using a Splitter for producing the single components, Message Router for routing the components to the processor, and an Aggregator to produce the resulting complex message.

Pattern	*Scatter-Gather (297)*
Pictogram (HW)	
Description	The Scatter-Gather pattern is intended for distributing a message to multiple recipients and waiting for a reply of each of them. The distribution might be a broadcast using a publish-subscribe channel, or via a recipient list allowing better control.
Realization	The realization is usage of the before-mentioned patterns; in this example of a recipient list and an aggregator. Fig. 27: Scatter-Gather using Recipient List and Aggregator

Pattern	*Routing Slip (301)*
Pictogram (HW)	

Description	A Routing Slip determines a chain of processes a message has to be routed through. Efficiency is in the focus of this pattern. The slip may be attached to the message or determined dynamically. Several options for routing are possible, where all options share the direct routing to the next processor. How many routers are used depends on the actual realization.
Realization	This complex pattern can be realized by using content-based or dynamic routers. The router decides, to which processor a message is routed to next, or if a processor should be bypassed.

Pattern	*Process Manger (312)*
Pictogram (HW)	
Description	A Process Manager is an extension to the Routing Slip pattern as it also processes a message in multiple processors, but determines more dynamically, which process to use next. This decision may also depend on the result of the previous processor.
Realization	Basically this pattern can be implemented with the patterns already introduced above. However the process manager has to keep track of the state of a message; that is, of the processing steps already applied to a message. Normally a process manager should be able to process many messages concurrently or provide multiple instances for processing.

Pattern	*Message Broker (322)*
Pictogram (HW)	

Description	Using a direct point-to-point channel for each communication between two components might lead to an *integration spaghetti* and thus in an unmanageable system concerning communication. A Message Broker acts as central hub for all message, thus allowing an easier manageable messaging infrastructure. There can also be a hierarchical order of Message Brokers.
Realization	The Message Broker is a complex system using mainly the different Message Router patterns shown above depending on the actual layout of the message channels. The channels themselves are more likely *Messages Buses*, thus the use of Point-to-Point Channels is avoided.

Message Translator A message translator or transformer converts a message into a data format expected by the corresponding receiver of this message.

Pattern	*Message Translator (85)*
Pictogram (HW)	
Description	As different applications should be connected, coherence of data types is not very likely. Different data types or at least different representation of data types are likely to occur in independently developed applications. Since changing applications is cumbersome and error-prone (and might introduce new difference for other applications), a message translator shall translate output of one application into input of a second, when this is needed. This is a simple form of an adapter (one transformation rule for complex data types). Transformation might be needed on the level of data structures, data types, data representation, or transport (message encapsulation).

Realization	As the Message Translator pattern works on the representation of data (on the four levels mentioned above), it corresponds to a transition taking one input format and providing a second, output format. The transition may be refined to allow a more detailed descriptions of the message translation. Depending on the input and output format, the function providing the translation can be bijective, thus we can reverse the translation. It should be at least injective, however we cannot always circumvent an information loss during translation.

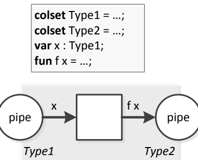


Fig. 28: Message Translator

Message Transformation

Pattern	*Envelope Wrapper (330)*
Pictogram (HW)	
Description	An Envelope Wrapper connects an application to a messaging system by wrapping data into a message complying to the requirements of the messaging System.
Realization	The Envelope Wrapper pattern is similar to a *Messaging Endpoint*. The Envelope Wrapper however focuses on just wrapping existing data into a system conforming message, where as an messaging endpoint may also handle further aspects as data layout and protocol.

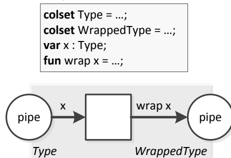


Fig. 29: Message Wrapper

Pattern	*Content Enricher (336)*

Pictogram (HW)

Description — It might be, that a sender of a message cannot provide all information needed by the recipient. Then a Content Enricher can be used to enrich a message with additional information using extra resources.

Realization — When realizing a Content Enricher, we have to take into account the source for additional information. For an incoming message we send a request to this external source and use the result to actually enrich the original message.

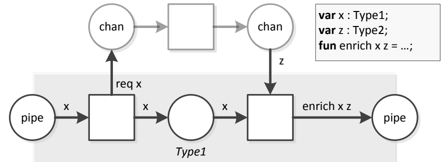


Fig. 30: Content Richer using additional information source

Pattern	*Content Filter (342)*

Pictogram (HW)

Description — The Content Filter removes parts of the message not required by the recipient, or it simplifies the data structure.

Realization — The Content Filter is a special variant of a *Message Translator* by not translating the whole message, but only the parts needed. If we consider a structured datatype, the Content Filter acts as projection on the parts we are actually interested in.

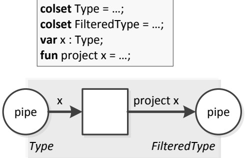
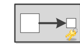


Fig. 31: Content Filter

Pattern	*Claim Check (346)*

Pictogram (HW)	
Description	The Claim Check pattern partially works like a content filter. Only relevant data should be provided for a certain claim check (e. g., for privacy reasons). However, later we need to correlate the reply to the original request again. Therefore we have to create an unique identifier being passed with the filtered claim and being stored with the original request.
Realization	In the Claim Check pattern we receive a claim to be checked. The filtering happens similar to the *Content Filter* pattern. Additionally the Claim type has to store a generated Key value. We also store the same value together with the original request in a data store, such that we can correlate the result of the claim check with the original request.

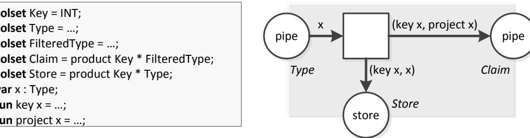


Fig. 32: Claim Check

Pattern	*Normalizer (352)*
Pictogram (HW)	
Description	A Normalizer is able to process messages with different formats and convert each message into a common format.
Realization	The Normalizer pattern is high-level concept. We can realize it as combination of a message router and different message translators. The router selects an appropriate translator to translate a message into the common format.

Pattern	*Canonical Data Model (355)*
Pictogram (HW)	
Description	Using a Canonical Data Model simplifies the central messaging infrastructure as only one data model has to be considered. However each application has to use the Canonical Data Model or a message translator is needed.

Realization	Given the concept of a Normalizer as above, it is technically manageable to use a Canonical Data Model.

Message Endpoint Message Endpoints form the interface to the messaging system. Often they are part of an additional layer for an application not aware of messaging.

Pattern	*Message Endpoint (95)*
Pictogram (HW)	
Description	In order to allow an application to use a messaging system, we need an API for connecting the application to the messaging system. A Message Endpoint therefore allows an application to create a valid message carrying data to be transmitted and actually to send the message (or receive it). It is the actual interface between an application and one message channel. It is a special form of a channel adapter and should act as a messaging gateway.
Realization	The realization is very specific to the application and to the channel used for messaging. For both sides the implementation is individual. In the simplest case, it looks like a message translator.

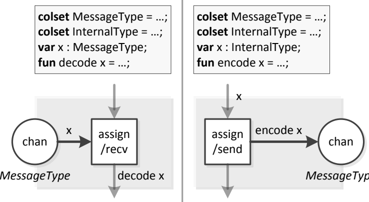


Fig. 33: Inbound and outbound Message Endpoint

Figure 33 above shows an incoming and an outgoing Message Endpoint. The idea is that every reading access to global variable x actually means receiving message x from an inbound endpoint, and every assignment to global variable x triggers the sending of the newly assigned content as message via an outbound endpoint.

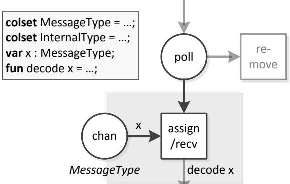
Messaging Endpoints The following patterns are special forms of a Messaging Endpoint, because they specify they purpose as an endpoint, especially regarding behavior.

Pattern	*Messaging Gateway (468)*
Pictogram (HW)	
Description	This pattern shall stronger encapsulate the messaging functionality. Since most messaging API provide similar functionality, a Messaging Gateway shall hide the actual message calls, especially the asynchronous exchange, and provide a generic interface for messaging.
Realization	The Messaging Gateway is a special form of a Messaging Endpoint. Its actual realization highly depends on the context, but it should resemble the above pattern.

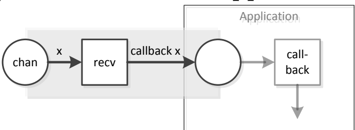
Pattern	*Messaging Mapper (477)*
Pictogram (HW)	
Description	A Messaging Mapper shall translate domain object into message and vice-versa, while keeping both separated. Neither type knows about the other. In contrast to a *Message Translator*, the mapper has not only to handle the structural translation, but it also has to provide a translation for object references or data types.
Realization	The realization is the actual translation of a domain object and back. The Messaging Mapper exploits domain knowledge to make this efficient.

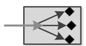
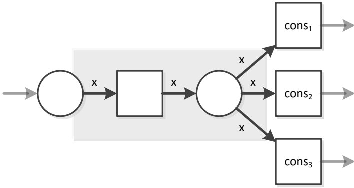
Pattern	*Transactional Client (484)*
Pictogram (HW)	
Description	Although a messaging systems has transactional aspects by itself (e. g., delivery of messages), a messaging client may be allowed to control the level of transactions.

Realization	As useful as this concept is, we cannot give a generic realization here, as the pattern highly depends on which actions in messaging system should actually be combined in a transaction.

Pattern	*Polling Consumer (494)*
Pictogram (HW)	
Description	A Polling Consumer decides, when to actually receive a message by making an explicit poll on the channel. When the call is made and no message present, then the consumer blocks until the message can be received.
Realization	The behavior of a blocking receive is for free in Petri Nets, as a transition can only fire, if all needed tokens are present. A missing message hinders a receiving transition from firing and thus blocks the application.  Fig. 34: Polling Consumer If the application does not want to wait forever for a message, it can remove the token from the `poll` place again, maybe based on some timeout.

Pattern	*Event-Driven Consumer (498)*
Pictogram (HW)	
Description	Since the *Polling Consumer* blocks, when it cannot receive a message, as an alternative a receiver can register an Event-Driven Consumer with the messaging system, which makes a callback to the receiver, when a message arrives.

Realization	The Event-Driven Consumer reacts on message arrival and triggers a callback in the actual application. Accordingly the Petri Net fires a transition on arrival of a message and forwards a token with a callback value to the corresponding callback point of the actual application.  Fig. 35: Event-Driven Consumer

Pattern	*Competing Consumers (502)*
Pictogram (HW)	
Description	The consumption of messages from a channel by one application may prove as bottleneck in messaging system. We may want to receive the messages concurrently by several consumers.
Realization	The realization is similar to a *Point-to-Point Channel* with multiple receivers. These receivers also concurrently try to consume messages from the channel, thus speeding up overall consumption of messages.  Fig. 36: Competing Consumers In which order the competing consumers receive a message is a matter of implementation. Any scheduling may be appropriate here, or the consumers may receive non-deterministically messages.

Pattern	*Message Dispatcher (508)*
Pictogram (HW)	

Description	A Message Dispatcher acts as mediator for incoming messages and distributes these message to one of a given set of consumers.
Realization	A Message Dispatcher is similar to a *Message Router*. However, the Message Dispatcher acts on a different layer, namely right on the channel. In the realization as CPN they are also similar. Additional schedules like round-robin for dispatching messages are also possible.

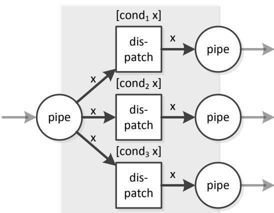
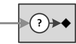


Fig. 37: Message Dispatcher

Pattern	*Selective Consumer (515)*
Pictogram (HW)	
Description	A Selective Consumer does not consume every message on a channel, but selects the messages it wants to consume. Every message not being consumed remains on the channel.
Realization	The realization is similar to a *Message Filter*. Whereas a Message Filter drops a message not meeting the filter, a Selective Consumer simply does not consume the message, thus only forwarding messages meeting the filter criterion.

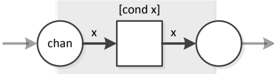


Fig. 38: Selective Consumer

Pattern	*Durable Subscriber (522)*
Pictogram (HW)	

Description	From a *Publish-Subscribe Channel* an application normally only receives messages as long as it is connected to channel. When disconnecting because of maintenance or similar reasons, the application may miss important messages. A Durable Subscriber still receives these message although the actual application is momentarily disconnected. When the application comes back online, the Durable Subscriber can forward the messages to the application.
Realization	This pattern is interesting in a system with changing participants and thus a changing infrastructure. We cannot cope with such changes in Petri Nets.

Pattern	*Idempotent Receiver (528)*
Pictogram (HW)	
Description	In a system with unreliable channels, a sender may have to send a messages multiple times before receiving an acknowledgment. Thus it might happen, that the receiver gets a message twice, and it has to handle the duplicates. An Idempotent Receiver shall show the same effect, whether receiving a message once or multiple times; either by explicitly removing duplicates or by defining corresponding message semantics.
Realization	The more practical solution of both alternatives is to store received messages.

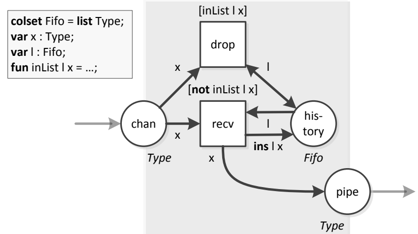


Fig. 39: Storing messages for an Idempotent Receiver

We keep each message—or alternatively an unique identifier of this message —in a list on place history. When a new message arrives, which has already been seen, transition drop can fire, as indicated by the guard, and effectively discards the message. If the message has not been received before, then transition recv can fire, because message x is not yet in the list. By firing recv the list is update by inserting the received message.

The practical limitation of this approach is the length of the list; thus the number of messages to remember. We would restrict the length of the list in order to allow a better performance and risking the unlikely event, that a duplicate message arrives with a huge delay.

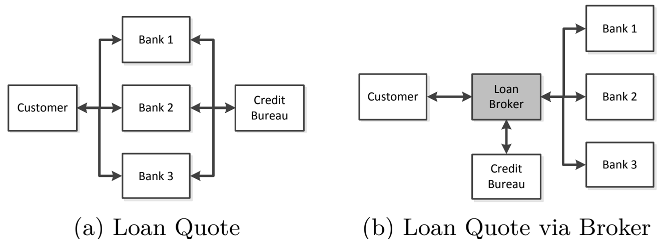
Pattern	*Service Activator (532)*
Pictogram (HW)	
Description	A service application is fixed to the communication technique provided by the implementation language. As a communication partner might use a different technique, we may not want to change the service application, but we provide a Service Activator connecting the messaging channel to the application. This way we decouple the reception of a message and the decoding for the application.
Realization	The realization is a *Polling Consumer* or an *Event-Driven Consumer*, analyzing an incoming message and invoking the appropriate service application.

3 Examples

We now show two examples for the application of CPN patterns. We first show how to use Enterprise Integration Patterns to model a system, and then we substitute the corresponding CPN patterns. The connection points between two patterns are either pipes or channels—in the Enterprise Integration Patterns as well as in CPN patters.

3.1 Loan Broker

Following we consider an example of a loan broker (cf. [1, page 361]). As Fig. 40 shows on the left. A customer wants to get a good loan quote. The customer asks several banks for choosing the best. In order to present an appropriate loan quote, each bank checks with a credit bureau for the customer's past.



(a) Loan Quote (b) Loan Quote via Broker

Fig. 40: Getting a loan quote without and with loan broker

On the right side of Fig. 40, we introduce a loan broker as central component in the system. The loan broker centrally manages the customer's request, receives the customer's credit history from the Credit Bureau only once and then asks three banks for a loan quote. The best offer is sent to the customer.

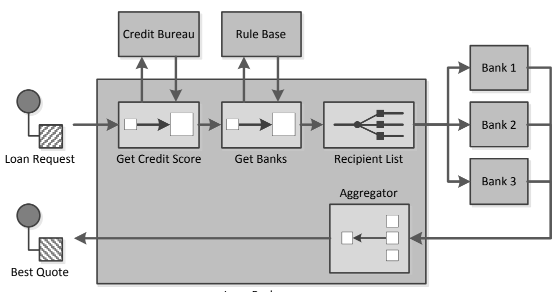


Fig. 41: Design of loan broker

Figure 41 shows a simple design for the loan broker using the patterns. Receiving a *loan request*, the request is *enriched with the credit score* of the Credit Bureau. A Rule Base further *enriches the request with potential banks* (some banks might be already excluded for known conditions), and the recipient banks receive the enriched loan request. For all banks, there is only a single return channel. An *Aggregator* receives all the replies and chooses the best offer, which is forward to the customer as *Best Quote*.

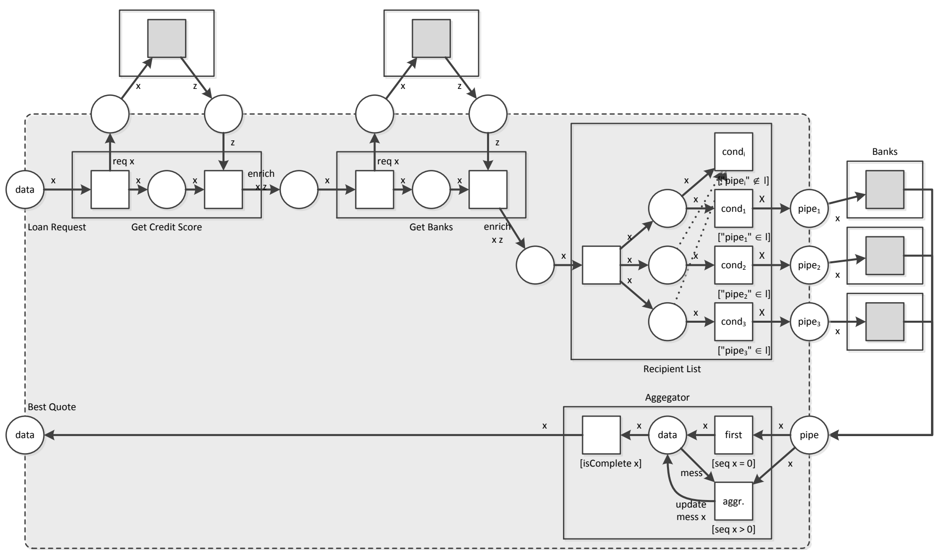


Fig. 42: Loan broker as Petri net

Figure 42 shows a Petri net implementation of the design proposed in Fig. 41 using the CPN patterns defined in Sect. 2. *Loan Request* and *Best Quote* are data types, thus we have corresponding places. The Content Enricher *Get Credit Score* and *Get Banks* make an RPC call to the Credit Bureau and the Rule Base (resp.) and enrich the Loan Request with the results. The recipient list shall distribute the request to the banks based on results of the rule base. The banks send back the result, and the *Aggregator* choose one of the incoming answers (the best one).

Figure 43 shows the realization of the loan broker example in CPN Tools allowing simulation and analysis.

Although the data flow dictates the control flow, the order of execution of each pattern is not determined to the very end. Depending on the number of messages send via the recipient list, the aggregator has to wait for the same number of messages. Somehow, this information has to be transported from the recipient list to the aggregator. Possible solutions might either be explicitly modeling additional data flow, but also controller synthesis for the actions of the Loan Broker may help to overcome this problem.

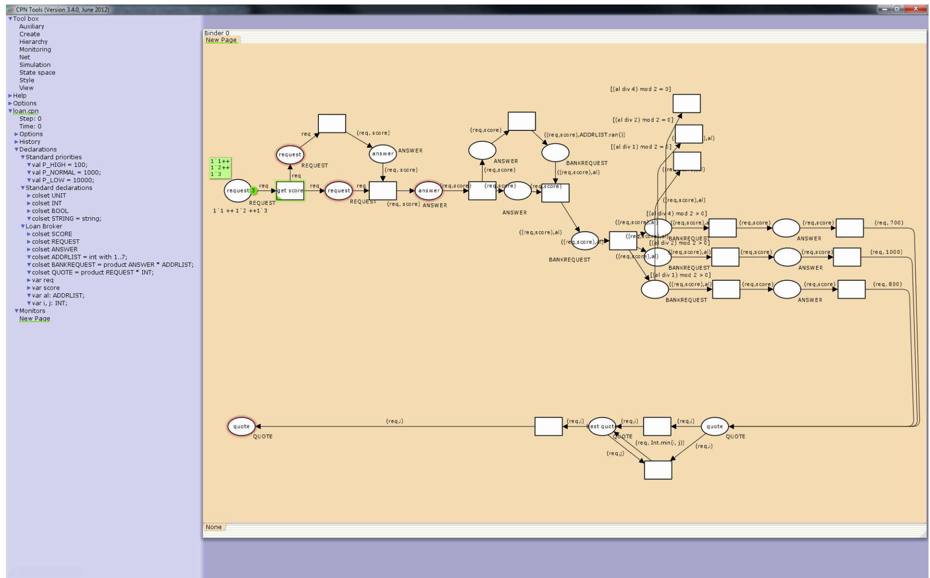


Fig. 43: Loan broker example in CPN Tools

3.2 Adapter Synthesis based on Patterns

In this example we exploit Enterprise Integration Patterns to *adapt two web services*. A web service is a special kind of application encapsulated behind a well-defined interface and intended for loose coupling. When we want to connect two such web services being developed independently, it is likely that the message flow between both cannot be matched exactly.

We shortly sketch now the basic idea for synthesizing an *adapter* to overcome this mismatch. We then explain on an example how use patterns to describe the message flow between two services and how the CPN realization allows us to synthesize an adapter.

We consider Google's Checkout [6] payment service and a proprietary payment back-end of a web shop to be adapted. As Fig. 44 indicates, we want to use patterns to model a mediator directing messages between both services. However, to ensure correct behavior for the service, we have to synthesize a controller taking care of the correct application of patterns. The combination of controller and mediator we call adapter.

Adapter Synthesis in a Nutshell An *adapter* A between two services S_1 and S_2, in the context of Enterprise Integration Patterns, is a messaging system that connects the two services S_1 and S_2. Usually, A is designed so that the composed system of S_1, A, and S_2 satisfies particular functional properties. For instance, the composed system never deadlocks.

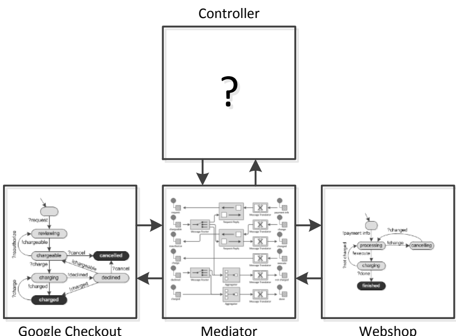


Fig. 44: A pattern based describes the message flow between two services. A automatically synthesized controller ensure proper application of EIP.

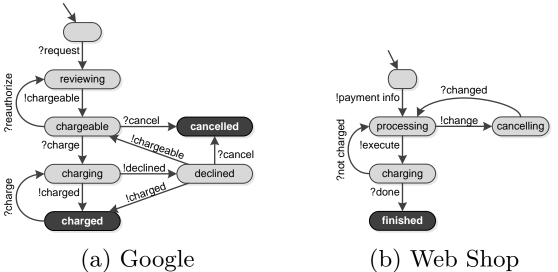
We follow the idea of [7] to obtain an adapter that yields a functionally correct system by separating message flow from message contents. In [7], the adapter A consists of two parts: a mediator M and a controller C. The mediator M defines message transformation rules and message routing rules; M can be completely designed using the Enterprise Integration Patterns introduced above. However, M alone is not sufficient to guarantee functional correctness: M might provide several transformation rules to transform a message by S_1 into different messages for S_2. Which message is required by S_2 may depend on the particular state of S_1 and S_2 at run-time. If the message is transformed wrongly by M, S_2 might received a message it is not expecting while waiting for a message it needs to continue. To prevent M from applying the wrong message transformation rule, the application of rules in M has to be controlled by the controller C that is local M. The composition of M and C is the adapter A.

For the property that the system consisting of S_1, A, and S_2 is deadlock free or weakly terminating (final state can always be reached), the controller C can be synthesized automatically, given M and abstract models of S_1 and S_2 [7,8].

Real-Life Example: Synthesizing Adapters for Google Checkout In the following, we show how the adapter synthesis technique of [7,8] can be used to construct an adapter between a proprietary web shop and Google's Checkout service [6]. We first show how to design a mediator between shop and Checkout service using the Enterprise Integration Patterns and then how to complete the mediator to an adapter by adapter synthesis.

The Google Checkout service and the web shop are *stateful* services; the behavior of each service is shown in Fig. 45. We assume that both components were developed independently and cannot be changed.

Google offers a service for sellers that can handle payment details as back-end of a web shop. We show it as automaton in Fig. 45a, where a gray box represent a state (black for a final state), an arc represents a transition, while an arc label means communication (! sending and ? receiving a message).



(a) Google (b) Web Shop

Fig. 45: Google Checkout [6] and Web Shop protocols

First Google Checkout needs a request to start. After the reviewing state the answer is chargeable. We left out the case, where this is not the case for facilitating the example. Then, the payment may be reauthorized, e. g., when payment details were changed, the whole payment may be cancelled, or it may receive a message to actually charge. Charging may be declined first, and then either fail as indicated by the message chargeable, or it may ultimately succeed, as indicated by a charged message. In the final state, a further charge message may trigger additional payment rates.

Let us assume, that we have developed our own web shop and implemented a payment back-end that we now want to connect to Google Checkout. As we can see in Fig. 45b, we first want to send the payment information. Then we either want to change some of the information, which is quit by a changed message, or we execute the payment. If the payment was not charged, we can change and execute again. If the payment is done, we are finished.

Please note: The depiction of the service does not indicate, whether communication is synchronous or asynchronous. Google's protocol expects synchronous invocation, whereas for the web shop we assume asynchronous communication.

If we look closely at both service, we can see that the messages sent and received by both services differ, as well as any behavioral relation of messages. It looks reasonable to relate the shop's payment info with Google's request. The execute should be related to Google's charge message, as well as the reply charged to the shop's done. However, Google's chargeable message is used in different contexts and needs more careful consideration.

For that reason we want to introduce a message mediator where we model the message flow as we expect it to be.

Mediator. We first describe the mediator on a higher level using Enterprise Integration Patterns and thus describing the intended message flow between services as shown in Fig. 46. Afterward we discuss the CPN realization in Fig. 47. Last we discuss control-flow issues and how they can be solved by controller synthesis.

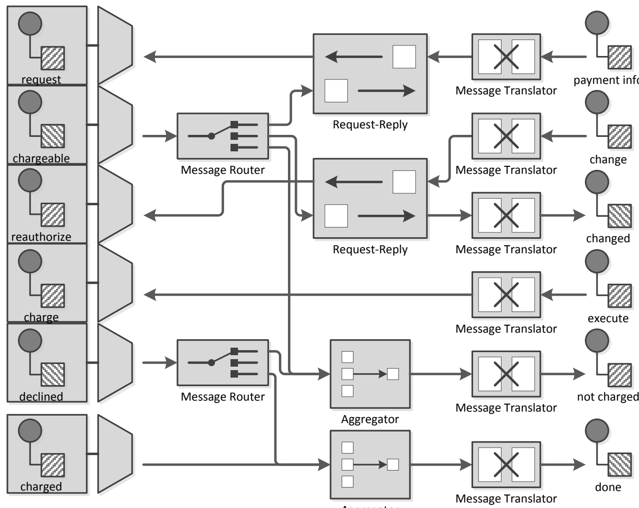


Fig. 46: Patterns-based mediator

Considering the different message types as well as the context each message type is used in, we decide to use the following patterns to describe the message flow.

The web shop's initial payment info triggers a *Request-Reply* pattern. Google Checkout receives a request and replies chargeable. The answer is not used by the web shop, but it is part of the protocol, so the mediator drops the reply.

Next, we have a *Request-Reply* pattern again. The change request is forwarded to Google Checkout as reauthorize message. The reply is a chargeable message again, which is forwarded as changed message to the web shop.

The execute message of the web shop is processed by a *Message Translator* and triggers the charge in Google Checkout. Each of the other message types also needs a message translator for connecting Google Checkout and the web shop. We decided to do this on the side of the web shop interface.

If an *Aggregator* receives a declined as well as a chargeable message, it sends a not charged message to the web shop, such that it can repeat the payment.

If a second *Aggregator* receives a declined and a charged message, or a charged message alone, it forwards a done to the webshop.

Since the messages chargeable and declined may be used by several different patterns, we introduce for each message type a *Message Router* determining where to a message should be forwarded.

Google's Checkout service uses synchronous message transfer. For each message type we add a *Channel Adapter* to translate the communication to asynchronous calls.

The patterns allow us to adequately model the message flow between both services. In order to connect them, we have to implement the used patterns and thus are able to adapt both services instead of changing them.

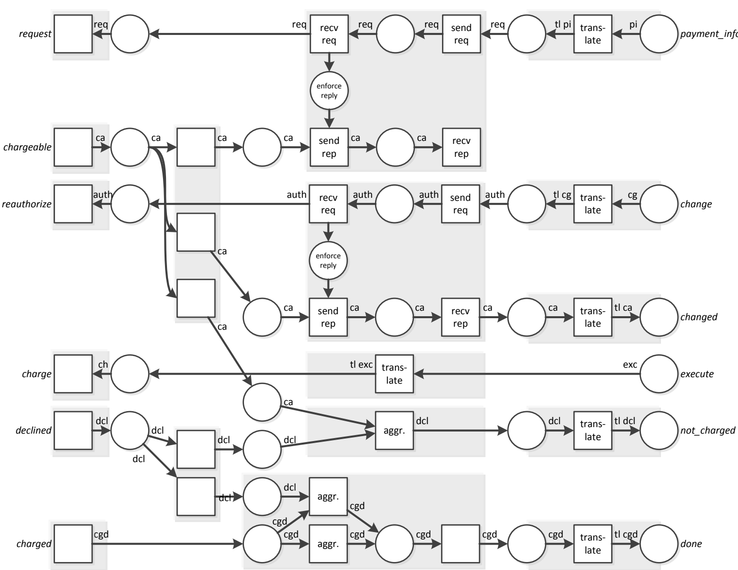


Fig. 47: Mediator

The implementation as Coloured Petri net is depicted in Fig. 47. For each pattern in Fig. 46 we used the pattern realization presented above.

Given the mediator's CPN model, we have to consider how to provide the transition conditions included in some of the patterns. Although in general such conditions may be provided, there are many cases, where we cannot decide locally for one transition, when it should fire. Since a condition always only knows its local context, providing a condition then is impossible and we need a more global point of view and thus a controlling instance.

For example, let us have a closer look at the message type and state called chargeable. The Checkout service can reach this state after receiving the initial request, after the task to reauthorize a payment, or after it declined a payment. The mediator just receives the chargeable message without any context information, and the corresponding *Message Router* has to pick the right pattern to forward the message to.

As it is essential, which message has been exchanged between Google Checkout and Mediator before receiving chargeable, we do not need local conditions for the message mediator, but we need control-flow relations between the patterns. When a request has been sent to Google Checkout, then chargeable has to be forwarded to the first *Request-Reply* pattern. If it was a reauthorize, then to the

second Request-Reply. If a declined preceded the chargeable, then it has to be forwarded to the first *Aggregator* pattern.

Something similar happens with the declined message. At the point, when the message arrives at the mediator, it is not clear, whether a chargeable or a charged will follow. Thus the corresponding *Message Router* cannot decide locally, which message will come next, such that we need control-flow dependencies again.

For the second *Aggregator*, we cannot decide locally, whether the first or second case occurs; that is, whether the charged occurs without or with a declined.

In any of these case, the result will be, that the mediator might be sending a message to the web shop that it does not expect. While the message itself may be ignored, the message expected by the web shop is actually missing such that it cannot continue. So if a chargeable message should be routed to the web shop as changed message, but instead is forwarded to the first *Request-Reply* pattern and simply dropped, than the web shop will wait for ever for the changed message and it will deadlock.

Nevertheless, we suggest to concentrate only on the message flow when designing the mediator as the control flow dependencies can be synthesized automatically.

Adapter Synthesis Following the idea to reduce adapter synthesis to controller synthesis, we can declare each transition of the mediator as controllable and observable, thus a controller can influence and monitor the mediator's behavior.

We have shown the schematics of this idea in Fig. 44. We have three components as discussed before: the two services Google Checkout and the web shop, as well as the mediator that we developed based on EIP. The fourth component—the controller—shall now be synthesized automatically. The controller is only allowed to communicate with the mediator—as indicated by the arrows—because a service normally does not allow any further outside control.

In adapter synthesis we consider the behavior of the overall system of both services and mediator. We therefore have to compose these systems. The remaining interface consists of the transitions of the mediator. Controller synthesis now looks at the state space of the composed systems and only allows a mediator's transition to fire, if this does not lead to a bad state like a deadlock. The resulting controller guarantees correct behavior of the system with the mediator ensuring correct message flow. For further details on adapter synthesis, we refer to literature [7].

The resulting controller can be seen in Fig. 48. We have arranged the pieces of the figure such that the controller's transitions are located where we have seen their mediator's counterparts. Additionally we have grouped transitions synchronizing with transitions belonging to one particular pattern with a gray box. Each of the blue transitions synchronizes with one transition in the mediator. Please note, that there are split transitions, like in the first message router, where five cases for forwarding the chargeable message to the first aggregator are shown. This happens, because the controller wants to distinguish certain situations; that is, different states of the two given services while firing the cor-

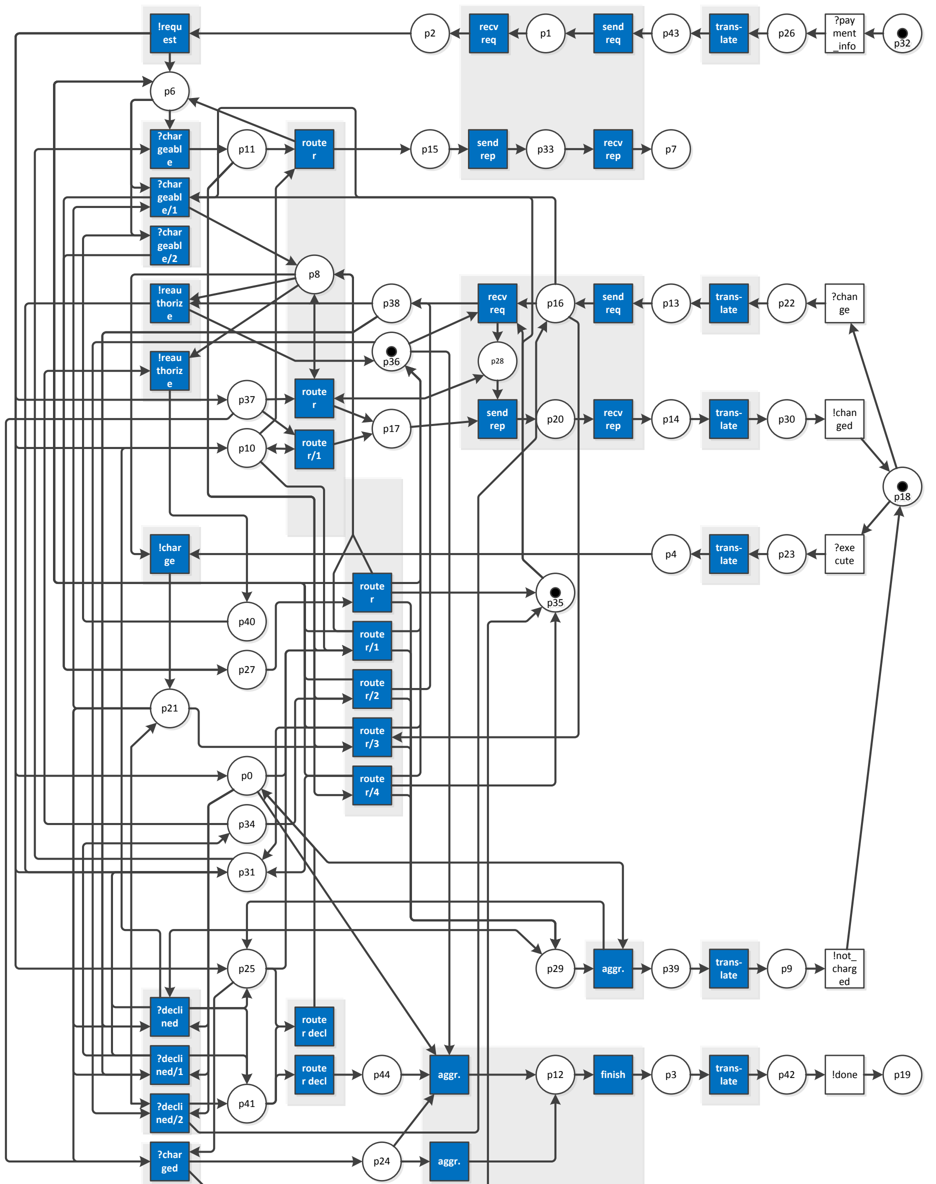


Fig. 48: Synthesized Controller

responding transition in the mediator. This also results in complex dependencies which manifest in the shown places and arcs.

The white transitions on the right-hand side are the result of the adapter approach. The adapter engine explicitly sends or receives a message, when exchanging messages asynchronously. These transitions realize this behavior. As an effect, the controller is able to react on the reception of a message directly before applying any pattern.

We can now use the mediator as blueprint for implementation, while the controller dictates the order of execution of the mediator's transitions.

4 Concluding Remarks

Enterprise Integration Patterns are a set of widely used patterns allowing a structured and efficient implementation of a messaging system. For these patterns we have provided a realization as Petri nets; thus each pattern is a Petri net block that can be connected to other blocks as indicated by the used patterns.

It is quite unlikely that a real implementation of a messaging system is done with Petri nets. However, Petri nets provide a wide range of analysis tasks, where the original patterns are to abstract and the actual implementation is to specific. Especially for model checking [9] several techniques for Petri nets exist, whereas model checking of software components is only solved for special cases (and in general not possible).

References

1. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
2. OMG: Business Process Model and Notation (2011) http://www.omg.org/spec/BPMN/2.0/PDF/ [retrieved on Oct 19, 2012].
3. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51
4. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
5. CPN Tools: http://cpntools.org/documentation/start [retrieved on Sep 21, 2012].
6. Google: Checkout https://checkout.google.com/ [retrieved on Oct 19, 2012].
7. Gierds, C., Mooij, A.J., Wolf, K.: Reducing adapter synthesis to controller synthesis. IEEE Transactions on Services Computing **5** (2012) 72–85
8. Wolf, K.: Does my service have partners? T. Petri Nets and Other Models of Concurrency **2** (2009) 152–171
9. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)

List of all patterns

Aggregator (268), 23

Canonical Data Model (355), 31
Channel Adapter (127), 10
Claim Check (346), 30
Command Message (145), 13
Competing Consumers (502), 35
Composed Message Processor (294), 26
Content Enricher (336), 29
Content Filter (342), 30
Content-Based Router (230), 20
Correlation Identifier (163), 16

Datatype Channel (111), 8
Dead Letter Channel (119), 9
Document Message (147), 13
Durable Subscriber (522), 36
Dynamic Router (243), 21

Envelope Wrapper (330), 29
Event Message (151), 14
Event-Driven Consumer (498), 34

Format Indicator (180), 18

Guaranteed Delivery (122), 9

Idempotent Receiver (528), 37
Invalid Message Channel (115), 9

Message (66), 12
Message Broker (322), 27
Message Bus (137), 11
Message Channel (60), 5
Message Dispatcher (508), 35
Message Endpoint (95), 32
Message Expiration (176), 17
Message Filter (237), 21
Message Router (78), 19
Message Sequence (170), 17
Message Translator (85), 28
Messaging Bridge (133), 11
Messaging Gateway (468), 33
Messaging Mapper (477), 33

Normalizer (352), 31

Pipes and Filters (70), 18
Point-to-Point Channel (103), 5
Polling Consumer (494), 34
Process Manger (312), 27
Publish-Subscribe Channel (106), 7

Recipient List (249), 22
Request-Reply (154), 14
Resequencer (283), 24
Return Address (159), 15
Routing Slip (301), 26

Scatter-Gather (297), 26
Selective Consumer (515), 36
Service Activator (532), 38
Splitter (259), 22

Transactional Client (484), 33