Analyzing and Completing Middleware Designs for Enterprise Integration using Coloured Petri Nets

Dirk Fahland[1] and Christian Gierds[2]

[1] Technische Universiteit Eindhoven, the Netherlands, d.fahland@tue.nl
[2] Humboldt-Universität zu Berlin, Department of Computer Science, Germany, gierds@informatik.hu-berlin.de

Abstract. *Enterprise Integration Patterns* allow us to design a middleware system conceptually before actually implementing it. So far, the in-depth analysis of such a design was not feasible, as these patterns are only described informally. We introduce a translation of each of these patterns into a *Coloured Petri Net*, which allows to investigate and improve middleware system designs in early stages of development in a number of use cases, including validation and performance analysis using simulation, automatic completion of control-flow in middleware designs, verifying a design for errors and functional properties, and obtaining an implementation in automatic way.

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

1 Introduction

Information systems are often not built from scratch, but are the result of integrating existing software components and services into a larger system by means of *middleware* that connects the existing components. This very task of integrating preexisting components is challenging, especially if components of multiple parties shall be integrated. Conflicting interests and goals need to be resolved, and whether a particular integration solution is "good" is subject to numerous criteria. In a recent survey [17], SAP listed as goals the ability to design fast and correct middleware systems, to enable enterprise interoperability, and to monitor and continuously optimize existing integrations.

Hohpe and Woolf proposed a collection of *Enterprise Integration Patterns* (EIP) [24] to help addressing many of the challenges at early stages of an integration project. Each pattern in their collection encapsulates a key functionality typically found in middleware solutions, such as message creation, routing, filtering, etc. These patterns can then be used to abstractly describe complex middleware systems in a comprehensive manner. Ideally, once the design is complete, the middleware can be implemented in appropriate technology [24, Chap. 13]. However, the functionality and behavior of each pattern in the EIP-collection is only given in informal text. Thus, analyzing the design for functional errors, missing or incomplete functionality, or performance problems requires additional and non-trivial work. Likewise the step from an informal design to a working implementation is costly and subject to mistakes.

In this paper we address the open problem of *automatically* generating from an informal EIP-middleware design a formal model of the middleware that allows for (1) verifying the middleware design for functional errors, e.g., using model checking, (2) completing the middleware design in case of missing functionality, e.g., using controller synthesis techniques, (3) analyzing the performance of the designed system, e.g., using comprehensive simulation techniques, (4) automatically creating a running implementation of the middleware, e.g., using process engines or code generation.

Contribution. We provide for each pattern in the EIP-collection a formal model in terms of *Coloured Petri Nets* (CPN) [26], an extension of Petri nets that also describe the processing and exchange of typed data. Using this formalization, a given EIP-middleware design can be translated to a CPN model that describes the functionality of the middleware. For the CPN model in turn a variety of techniques are available: the CPN model can be modelchecked for functional errors [29], its control-flow can be completed using controller synthesis techniques [19], its performance can be analyzed using simulation techniques [26, 27], and an implementation of the middleware could be obtained by code generation [30] or by deploying the CPN model in a process engine [33].

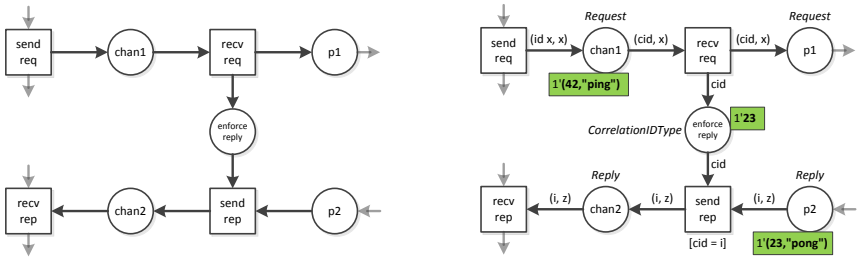
Outline. We give an overview on existing works on enterprise integration in Sect. 2. Then, we recall some basic notions of Coloured Petri nets in Sect. 3. Section 4 recalls the Enterprise Integration Patterns and we present the principles of formalizing EIP in CPN models for several comprehensive examples; the complete collection of formal patterns is available at [15]. We then show how to derive formal CPN models of a middleware design using our approach, and discuss in Sect. 5 how available techniques for verification, synthesis, simulation, and deployment of CPN models can be used in the context of middleware designs. We conclude in Sect. 6.

2 Related Work

Enterprise integration receives much attention from industry and academia yielding a large body of literature. Some recent surveys can be found at [9, 22, 39]. In the following we focus on approaches that address the use cases raised in Sect. 1.

Many authors, such as Scheibler and Leymann [41], advocate a pattern-based or model-driven approach for enterprise integration and many solutions have been proposed in this direction. Frantz et al. [16] introduce the DSL *Guaraná* that uses EIP to model enterprise application integration solutions and can be translated to Java code. *Spring Integration* [28] and *Apache Camel* [25] are further frameworks for modeling messaging systems based on EIP. Recent approaches for pattern languages [23, 34] concentrate on identifying patterns and implementing them, though missing the chance of giving formal semantics. The existence of such tools shows the acceptance of EIP, but these frameworks lack formal analysis facilities required in early stages of the design.

Approaches with formal semantics, among many others, propose AI planning [35], theory building [4], or model-driven architectures [38] to obtain a formal model of an



(a) Basic structure (b) CPN model

Fig. 1: Example of structure and inscriptions of a CPN

enterprise integration solution. However, none of the resulting models can address all the uses cases of Sect. 1. The perhaps most versatile approach thus far is proposed by Mendes et al. [36] who show the feasibility of Petri Nets in Enterprise Integration. Unfortunately, their approach lacks a complete and structured support for all known Enterprise Integration Patterns and focuses on simulation and execution, only. That complex systems can be built from formalized patterns has been successfully demonstrated in other domains such as the *Workflow Patterns* [3] or Service Interaction Patterns [5].

To the best of our knowledge, the combination of EIP and a translation of each pattern into a formal model, such as Coloured Petri nets, for interweaving the worlds of pattern-based enterprise integration and versatile formal analysis has not been addressed before.

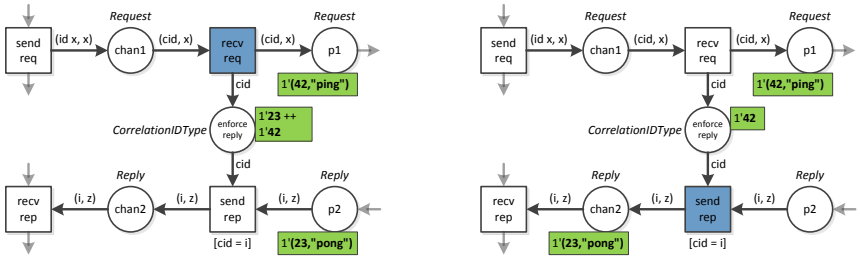
3 Preliminaries

We use *Coloured Petri Nets* (CPN) [26] to model the semantics of Enterprise Integration Patterns. CPN are successfully applied in research and industry in modeling and analyzing distributed systems [43].

A Petri Net processes resources called *token*. *Places* hold these resources, and *transitions* process them. A *flow relation* connects places with transitions and vice versa.

Figure 1a shows an example for the structure of a CPN; a circle depicts a place, a rectangle depicts a transition, and the arcs depict the flow relation. This basic structure already dictates that transition recv req has to consume a resource from place chan1 and that it produces new resources on places p1 and enforce reply.

In CPN each token is a value (called *color*) of some type (called *colorset*). Each place is typed with a specific colorset and holds only tokens of that type. In Fig. 1b places chan1 and p1 have type *Request*, enforce reply has type *CorrelationIDType*, and p2 and chan2 have type *Reply*. Each arc is labeled with either a variable (e.g., cid or x), or a complex term such as function applications (e.g., id x applying function id on variable x) or complex data structures (e.g., tuple (cid, x)). The labels of arcs adjacent to a transition express which tokens the transition consumes and produces as explained below. In addition, a transition can have a guard to restrict consumption and production of tokens; e.g., transition send rep has the guard [cid = i].



(a) After firing recv req

(b) After firing send rep

Fig. 2: Behavior of CPN: Effect of subsequent firing of recv req and send rep

The state of a CPN is a *marking* describing a distribution of tokens (colors) over places of the respective colorset. In Fig. 1b, place chan1 holds token (42,"ping") (a tuple of the colors 42 and "ping"), place enforce reply holds token 23, and place p2 holds token (23,"pong"),

The behavior of a CPN is described by *firing* transitions, which consume and produce tokens as follows. Figure 2a shows the effect of firing transition recv req at the marking of Fig. 1b. For firing recv req, the variables cid and x at the arcs adjacent to recv req have to be bound to a color, such for each incoming arc $(p, \text{recv req})$ its arc label evaluates to a token on place p; in this case recv req is *enabled*. For instance, for the binding cid $=$ 42 and x $=$ "ping", there is a token (42,"ping") on place chan1. If multiple bindings evaluate to available tokens, then one binding is chosen non-deterministically. When firing an enabled transition under the chosen binding, the tokens described by labels of incoming arcs are consumed from the respective place, and tokens described by labels of outgoing arcs are produced on the respective place. Firing recv req in our example consumes (42,"ping") from place chan1 and produces (42,"ping") on p1 and 42 on enforce reply. The resulting marking is shown in Fig. 2a.

In Fig. 2a, there is no binding to enable recv req because there is no token on chan1. Transition send rep still can fire for the binding i $=$ 23, x $=$ "pong", and cid $=$ 23. The guard of send rep ensures that i and cid are bound to the same value. The result of firing send rep is shown in Fig. 2b.

4 Enterprise Integration Patterns as Petri Nets

In their best practices book *Enterprise Integration Patterns* [24], Hohpe and Woolf have collected a widely used and accepted collection of integration patterns. The patterns 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.

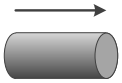
In the following, we give a short overview of typical EIP, explain their concepts and their CPN realization. The shown patterns (and CPNs) will be used later in examples

that illustrate several use cases. The complete list of patterns and CPN realizations is available at [15].

4.1 Basic Concepts of EIP

The following six patterns are the basic concepts described by Hohpe and Woolf; all further patterns are specializations.

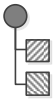
Message Channel



A *message channel* is the essential part of a messaging system. It determines, which applications are connected and ultimately how the applications communicate. Knowing how to direct a message to its destination is prerequisite for specifying message layout and manipulation.

Implementations of a message channel differ depending on its purpose (e.g., point-to-point or broadcast communication). Also technical aspects like buffer capacity of a channel or the order of messages influence the actual implementation. In any case, sending a message is normally decoupled from receiving a message, thus a channel needs the capability to hold a message. In CPN we realize this by using a place as buffer. In context of the intended properties, this place can be refined to a queue or to hold only a certain capacity of messages.

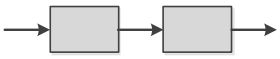
Message



A *message* is the atomic unit being transported through a message system. The main purpose of a message is to transfer information, either by encapsulating this information within the message or through the type of the message. Information within a message may be arbitrary complex. The type of a message also influences how a message is treated. We may explicitly send a command, a document, or a notification. Based on this categorization, a messaging system can transfer a message to an appropriate destination or handle a message with higher priority.

In CPN we represent a message by a colored token. The corresponding colorset determines the type of a message, and colorsets can be used to represent even complex data types.

Pipes and Filter

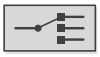


A *filter* is a message processing part of a messaging system. It allows to control, which message is forwarded to which recipient, and whether all parts of a message are forwarded or even additional information is added to a message. A *pipe* connects different message processing units and thus directs messages through the messaging system.

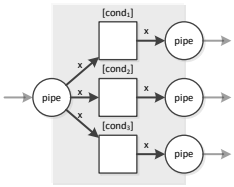
In this general form a filter can be represented as transition in CPN. It takes a message from a pipe and puts another message on a second pipe. The relation between the consumed and produced message is dictated by the filter's purpose and in general resembles function application.

A pipe has a similar purpose as a message channel. Therefore we use again a place to represent a pipe.

Message Router



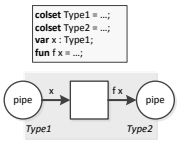
A *message router* is a special message filter. It takes an incoming message and directs it to one of many potential recipients. Which recipient a router picks may be decided statically or dynamically, based on rules or a message's content.



In CPN we can use a transition guard to decide whether a message should be forwarded via a certain pipe. The shape of the guard depends on the actual context of the message router. For instance, the guard may define a round-robin scheduler or use information contained in message x. We may even connect the transitions to a *rule* place providing additional dynamical information for picking the appropriate recipient.

Message Translator

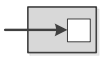
A *message translator* converts one message format into another. It is typically needed, when applications implement different message formats and we want these applications to exchange messages.



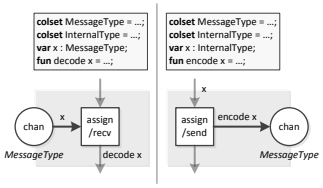
In CPN a message's format is defined by the colorset used for a token representing the message. The translator transition thus consumes a token of one type and provides a token of another type. We have to provide a function f which relates input and output and thus provides the message translation.

bijective. Especially, superfluous parts of the input format may be

Message Endpoint



A *message endpoint* connects an application with the messaging system. It allows the application to send or receive messages and thus exchange data. Within the application, a message endpoint might be hidden behind an additional layer allowing to use an application without fundamental changes.



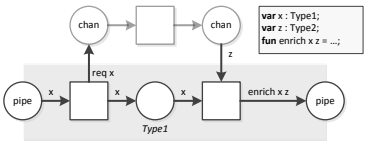
In CPN we either receive (left) or send (right) a message. We have to take care, that a message is appropriately encoded or decoded and thus the internal and a message's type are in a valid relation.

4.2 Detailed Description of Example EIP

We now present examples of more involved patterns often needed in practical integration scenarios; see [15] for the complete list. We have picked these patterns, because we will later use them in our examples and still they indicate the variety of these pattern.

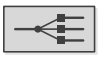
Content Enricher

A *content enricher* is a special form of a *message translator* and shall enrich a message with information needed by the recipient. This data might be added statically or dynamically by invoking another application that provides the required data.

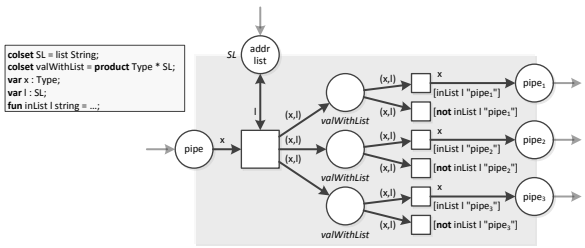


This pattern consumes a message x and produces an enriched version of x. If we want to add information statically, we can realize this pattern similar to the *message translator* pattern above. If we want to use an external application, we have to send a request req x based on the consumed message to that application. The application's answer z then is used to enrich the original message (enrich x z).

Recipient List



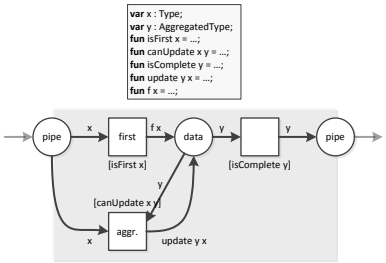
The *recipient list* refines the *message router* for multicast communication. A message is forwarded to multiple recipients at once; the set of recipients can be chosen dynamically from the set of all potential recipients.



When implementing a *recipient list*, we have to check, which recipient shall actually receive a message, and we have to provide a copy for each recipient. In CPN we first provide a copy for every potential recipient; the address list l contains the recipients that shall receive their copy. A message is then forwarded to a recipient if it is in the list (guard [inList l "pipe_i"]) and dropped otherwise (guard [not inList l "pipe_i"]).

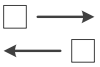
Aggregator

The *aggregator* is a kind of *filter* that consumes several messages which are aggregated into one message that is finally forwarded. The aggregated message may be a union of all received messages, or the one message that fits best to some criterion.

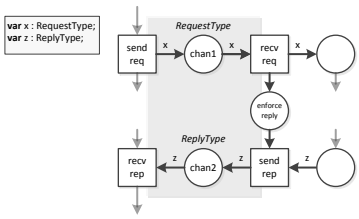


In the CPN model we put the first arriving message x on a data place (decided by guard [isFirst x]). Any further arriving message is used to update the stored message (update y x) if both can be aggregated (guard [canUpdate x y]) — either by keeping the best message or by joining both message. When all messages have arrived (condition [isComplete y]) we can forward the message.

Request-Reply

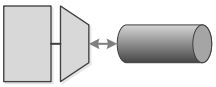


The *request-reply* pattern expresses a bidirectional exchange of messages. One service sends a request to a second one that has to provide a reply for the first one.

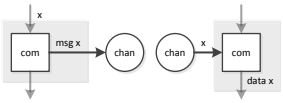


For realizing the request-reply pattern we need two channels chan1 and chan2—one for the request, the other for the reply. Correspondingly the services need transitions to send/receive the request/reply. When receiving the request, an additional token is produced on enforce reply for indicating that a reply is needed.

Channel Adapter



A *channel adapter* connects an application to a messaging channel. The channel adapter realizes sending or receiving messages by using an application's API or directly using and manipulating application data, so that the application does not have to be aware of message exchanges.



The pattern allows an application either to send or receive a message. What kind of data inside the application is affected by this is a matter of implementation. One prominent example for this pattern is the conversion of a synchronous message exchange into an asynchronous exchange, and vice versa.

Connecting patterns The mean to connect EIP with each other are the channel and pipe patterns that are represented by arcs. These arcs become places in the CPN model, such that the remaining patterns are connected via these places; i.e., one pattern will put a message on the place while the other takes a message from it. Please have a look at the example below (Fig. 4b).

While the EIP model does not necessarily need to distinguish the identity of arcs, in the CPN model the identity of places is more important. Consider the routing pattern, which has multiple outgoing pipes. If we provide conditions in the CPN model, then the right place should be marked if the condition comes true.

Connecting to EIP with an arc further implies type equality; i.e., the first pattern sends out a message of a certain type, then the second has to receive a message of the same type. Thus, in the CPN model the connecting place must have the same type. We actually can use this requirement to infer types when translating EIP to CPN.

The result of the translation from EIP to CPN results in a canonical net structure. However, as a user has to provide the definition of types and and functions, the translation is not unique, but depends on a user's data declarations.

4.3 Deriving a Formal Model of a Middleware using EIP

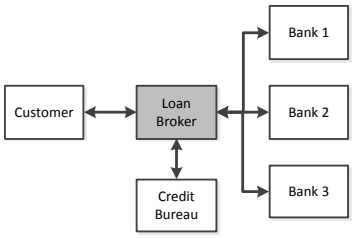


Fig. 3: Integrating a loan broker

We now show how to derive a formal model of a middleware design expressed in EIP using the CPN realizations introduced above. The given design provides several EIP having channels or pipes as their interfaces; arcs connect the different interfaces. The translation to CPN is then straight forward: replace each EIP with its CPN realization and merge two CPN places if they describe pipes/channels connected by an arc in the EIP design. The system's designer then has to provide the function definitions used in the patterns.

We want to implement a *loan broker*. The loan broker (Fig. 3) acts centrally between a customer, several banks, and a credit bureau for the customer's credit history.

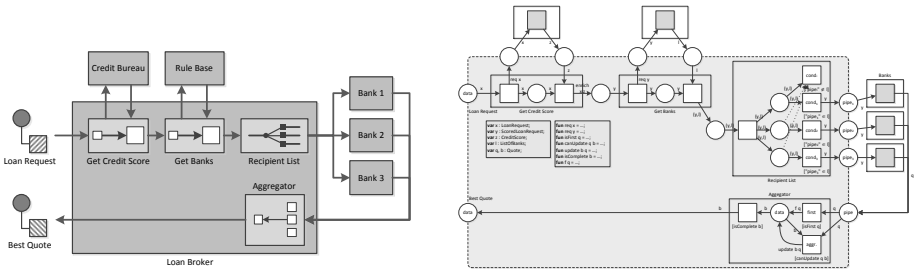
Figure 4a shows the loan broker's integration as proposed by Hohpe and Woolf. A customer's Loan Request first passes a *content enricher* to add the credit score provided by a Credit Bureau to the request. The next *content enricher* preselects addresses of some banks with the help of a rule base. A *recipient list* actually sends the request to some of three Banks. The results are handled by an *aggregator* and the Best Quote is returned to the customer.

Translating the EIP design of Fig. 4a as described above yields the CPN model shown in Fig. 4b. Such a formal CPN model of the middleware can then be used in various use cases as we discuss next.

5 Applications

Hohpe and Woolf assume that a *domain expert* uses EIP to model a messaging system and then implements each pattern. Even when taking great care, the EIP design may contain flaws or have performance problems which cannot be unveiled based on the informal description of EIP. In the following, we show for various use cases how the CPN realization of an EIP design helps discovering problems at early stages of design.

CPN offer a high level of abstraction while allowing local refinements. On this level, we can *simulate* and *model check* a CPN model as well as *create service adapters* or *run*



(a) EIP based (b) CPN based

Fig. 4: Loan broker integration

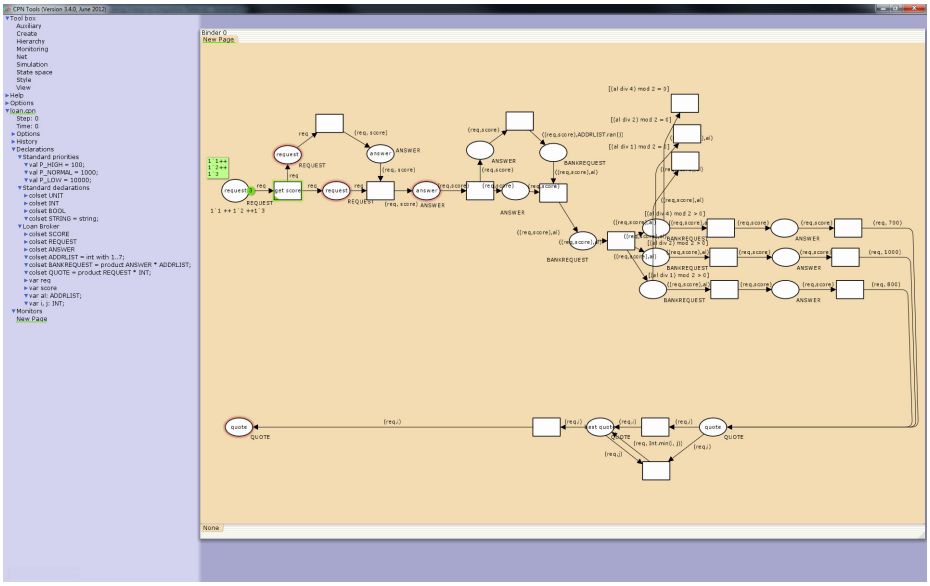


Fig. 5: Loan broker in CPN Tools

it on a workflow engine. Thus CPN models bridge the gap between conceptual models and their implementation. If required, the domain expert can use the CPN model as blueprint for implementation.

5.1 Simulation and Performance Analysis in CPN Tools

CPN have proved to be a useful tool to discover design problems and performance issues of complex distributed systems, documented by several dozen case studies with industrial context [43] (e.g., [11, 31] have a similar context as our loan broker example). Main focus in these case studies is the evaluation of CPN models before actually implementing the final system. This allows flexible modeling and changing, s.t. the final system will meet performance requirements.

CPN models have a formal semantics and thus can be executed. Missing implementation details manifest as abstraction or non-determinism. *CPN Tools* [27] run simulations on a CPN model yielding realistic analysis and performance data. With these data one can identify flaws and optimize the design before the actual implementation. This can particularly be done *during* the design phase allowing for an interactive analysis-driven design method.

Figure 5 shows the loan broker example in CPN Tools. Let us assume, the loan broker is paid on provision base for successfully procuring a loan quote. Given statistics on potential clients and offers of the banks, we can evaluate, how often a loan quote is offered and what the provision may be.

Since we already have the model at hands, even support for runtime decisions is possible. Rozinat et al. [40] developed a *simulation system for operational decision support*. They use YAWL [2] for running workflows and the ProM framework [1] for process mining and support decision making for the currently running workflow. Since

YAWL and CPN are closely related, this idea can also be applied for middleware systems that need guidance in complex interaction scenarios. The simulation results do not only allow to refine the messaging system before execution, but during execution simulation results allow to influence the message flow. The goal of distributing messages, s.t. the overall system load is low, can also be achieved by investigating multiple scenarios with simulation.

5.2 Model Checking

Simulation can be used to validate and improve a design, but is incapable of proving the absence of design errors. CPN Tools also allow to explore the state space of a CPN model [29] for the purpose of *verifying* that a particular property holds. The state space can be infinite (depending on data domains and the general net structure), however, verification is still feasible in many cases.

CPN Tools ship with an extension ASK-CTL [10, 12]. It implements a model checker allowing to check CPN models for temporal properties similar to the Computational Tree Logic (CTL) [13]. Alternatively, one may abstract from data aspects (by turning data-dependent decisions into non-deterministic choices) and consider only the control-flow and message flow of the designed middleware; the resulting net without any arc inscriptions or guards is called *Place/Transition net*. For Place/Transition nets a multitude of model checkers exist, allowing to verify temporal logic [21, 42], probabilistic properties [32], and timing constraints [18].

In our example of the loan broker, we may want to ask, if always a loan quote is sent back to the customer. There is the case, where the address list might be empty, s.t. no bank is contacted and therefore no loan quote offer arrives at the loan broker, which can be detected automatically including a counter example trace. With this knowledge, we can also ask for the probability of a loan quote, or how long it takes to find a best offer.

5.3 Automatically Completing Designs

Hohpe and Woolf focus on stateless applications (as in the loan broker example) that shall be connected by a messaging system, but EIP equally apply for integrating *stateful applications*. The difficulty arising here is that the integrating middleware may provide several message transformations that have to be applied in a *controlled* manner to avoid that the integrated system runs into a bad state, e.g., a deadlock. In the following, we show how a given EIP-based middleware design can be *automatically* completed to integrate stateful applications such that no error occurs.

We solve the problem by adapting a solution from the area of services, where a stateful application is called *service* and the integrating middleware a *service adapter* [44]. Several approaches propose to model the message flow between two services formally. The techniques cover the *Web Service Choreography Interface* language [8], *process algebra* [7], π*-calculus* [6] or the use of *message transformation rules* [19]. Dumas et al. [14] even propose a visual notation using rectangles with the name of the building block to use. Using the variety of EIP we can complement these techniques.

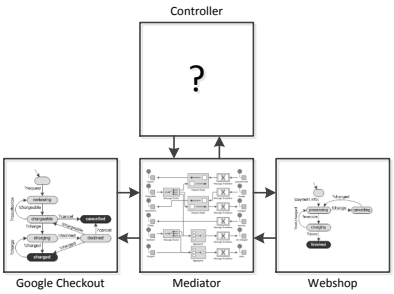


Fig. 6: Adapter synthesis

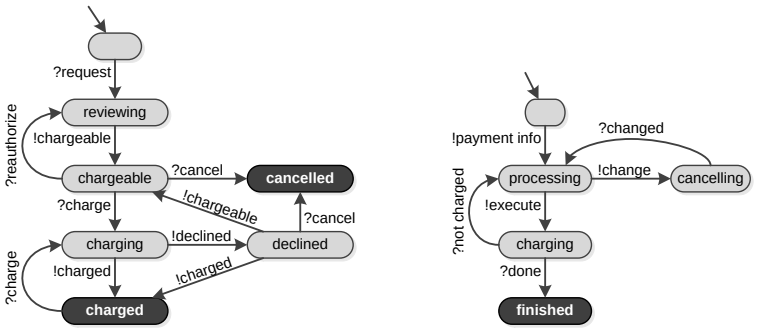
Based on the idea to *separate an adapter into an engine for the message flow and a controller* [19], we now show with the help of an example how to use EIP for modeling the message flow between two services and how we finally yield a complete and error-free service adapter. As Fig. 6 shows, the mediator connects to the given services (left and right), while a controller dictates the application of patterns in the mediator. Such a controller can be synthesized automatically, e.g., if we want deadlock-freedom in the system.

As example, we consider Google's Checkout payment service [20] and a proprietary web shop protocol (Fig. 7). Google Checkout is an API for handling web shop payments externally. We assume some proprietary payment back-end of such a web shop. We use communicating automata to describe the underlying protocols with gray boxes as states — black for final states — and arcs for transitions. An arc label tells us, whether the service sends (!) or receives (?) a certain message type. In Fig. 7 we do not distinguish the type of communication, but please note, that Google Checkout assumes *synchronous* message exchange, whereas the web shop uses *asynchronous* communication.

Google's Checkout service needs a request and information on the charge, each followed by sending an acknowledgment. We may also reauthorize a payment (e. g., by entering new credit card credentials), cancel the whole payment, or trigger a subsequent charge, if we transfer only parts of the whole sum. When a payment is declined at first, the service decides afterward, whether the payment was successfully charged or the payment has to be redone.

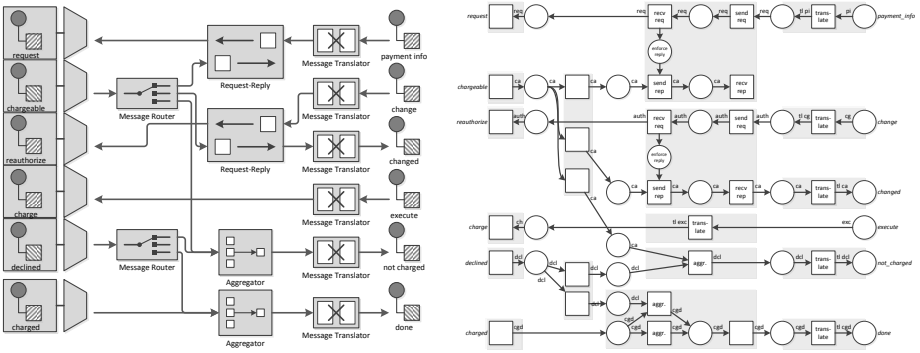
The web shop first sends the payment information and then either wants to change some of the information again, or it requests to execute the payment. If the payment is not done, then it returns to the processing state.

Besides the obvious mismatch in message names, we do not succeed in matching the interfaces of both services, s.t. they can communicate correctly; that is, that both



(a) Google Checkout (b) Web Shop

Fig. 7: Google Checkout [20] and Web Shop protocols



(a) Mediator (b) Engine

Fig. 8: Mediator and Engine for Google Checkout and Web Shop

services can *always* reach a final state together. We now first introduce a mediator using EIP to model the message flow between both services.

For building the mediator in Fig. 8a we consider the services' protocols: The web shop's payment info is *translated* into a request which is answered by a corresponding chargeable. The mediator uses a *request-reply* for message correlation and drops the reply that is not expected by the shop. When the web shop sends a change, then Checkout shall reauthorize the payment and quit this step with a chargeable — again the mediator uses the *request-reply* pattern. The execute message is directly translated into a charge. If Checkout's answer is a declined followed by chargeable, these two messages are *aggregated* to a not charged. A charged together with a possible declined is *aggregated* to a done. The *message routers* distribute the chargeable and declined messages, that are needed in different patterns. The *channel adapters* on Google Checkout's side adapt the synchronous communication of the service to the asynchronous nature of the patterns.

While the mediator design connects both, Checkout and web shop, it allows for erroneous runs that cannot be avoided. Whenever Checkout sends a chargeable message, the message router has to decide where the message shall be forwarded, but this decision cannot be made locally just in the router. For instance, to route chargeable to changed, the router has to know that a change message was sent earlier (but is no longer in the system). Without such knowledge, the message could be dropped (reply to payment info) and the system deadlocks, waiting for the reply on change. This can be prevented when the message router is controlled based on preceding message exchanges.

The engine service in Fig. 8b uses the CPN pattern to model the message flow and we can recognize the same structure as in the mediator. By giving the abstract service models of Google checkout and the web shop, and the engine model of Fig. 8b to the synthesis technique of [19], we automatically synthesize a controller that restricts the firing order of transitions, of the router as well as of all other transitions, such that the interaction is deadlock-free. For the given example, the synthesized controller has 212 states, and the resulting equivalent Petri net has 44 places, 41 transitions, and 138 arcs; the controller is shown in [15, p.47]. The synthesized controller can be implemented as independent component that accesses the interface of the engine.

5.4 Execution on Workflow Engine and Code Generation

Enterprise Integration Patterns are normally considered to be used for modeling the architecture of a messaging system, and then a developer has to implement the single patterns in languages like Java or C#. However, Coloured Petri Nets have a Turing-complete semantics, and thus are able to express any desired behavior expressible in any programing language.

Instead of implementing each pattern we can refine each CPN pattern and then directly execute the CPN model. Liu et al. [33] describe the *Tsinghua Workflow Management System* based on Coloured Petri Nets being suitable for this purpose. Although *YAWL* (Yet Another Workflow Language) [2] does not support the execution of arbitrary CPN, it is still based on CPN and shows the feasibility of executing CPN directly.

The CPN patterns may need some refinement in conditions or message translation, but these are rather small and local tasks and do not involve to implement a complete pattern in a programing language. Moreover changes in the model or the implementation are easily traceable, whereas a paradigm shift to a programing language usually ruins this kind of relation.

Implementations with better performance can also be obtained by generating equivalent code out of a CPN model [30, 37], though the particular constraints of middleware platforms may require adjustments of existing techniques.

6 Concluding Remarks

Enterprise integration will remain a hot topic, especially as industry needs to integrate existing applications into new infrastructures. Using EIP is a first step, because they allow to tackle the problem on a conceptual level and not only on the technical one.

The EIP by Hohpe and Woolf cover the large spectrum of concepts used in message-based middleware systems, for which we provided corresponding CPN pattern [15]. With the translation of EIP to CPN we provide the means to *analyze* and *improve* a system at early stages of its design using existing analysis techniques and tools. We discussed how the design can be checked for flaws and performance issues before the actual implementation and execution, and showed how to complete an incomplete design using controller synthesis techniques. This enables system designers to save time and cost, as necessary changes can be made early.

We expect, that the methodology used in this paper can be transferred to other contexts and thus enables analysis in context of other pattern collections. An aspect left open in this paper is to define appropriate functions for transforming and routing messages; such definitions could for instance be derived using semantic techniques.

We plan to integrate this approach into an editor that allows to model a middleware system using the Enterprise Integration Patterns. Additionally each instance of a pattern can be enriched by the information needed for the CPN pattern; i.e., message types, routing conditions, and so on. Then an automatic translation into a CPN model shall allow us to apply the setting presented in this paper.

References

1. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W.E., Weijters, A.J.M.M.: ProM 4.0: Comprehensive support for *real* process analysis. In: ICATPN. pp. 484–494 (2007)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. 30(4), 245–275 (2005)
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), 5–51 (2003)
4. Barn, B., Clark, T.: Revisiting Naur's programming as theory building for enterprise architecture modelling. In: Mouratidis, H., Rolland, C. (eds.) Advanced Information Systems Engineering, Lecture Notes in Computer Science, vol. 6741, pp. 229–236. Springer Berlin Heidelberg (2011)
5. Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Service interaction patterns. In: Business Process Management. pp. 302–318 (2005)
6. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software 74(1), 45–54 (2005)
7. Brogi, A., Canal, C., Pimentel, E.: On the semantics of software adaptation. Sci. Comput. Program. 61(2), 136–151 (2006)
8. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing web service choreographies. Electr. Notes Theor. Comput. Sci. 105, 73–94 (2004)
9. Chen, D., Doumeingts, G., Vernadat, F.: Architectures for enterprise integration and interoperability: Past, present and future. Computers in Industry 59(7), 647 – 659 (2008)
10. Cheng, A., Christensen, S., Mortensen, K.H.: Model checking Coloured Petri Nets - exploiting strongly connected components. Tech. rep., University of Aarhus (1996)
11. Cherkasova, L., Kotov, V.E., Rokicki, T.: On net modeling of industrial size concurrent systems. In: Application and Theory of Petri Nets. pp. 552–561 (1993)
12. Christensen, S., Mortensen, K.H.: ASK-CTL. University of Aarhus (1996)
13. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
14. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: Business Process Management. pp. 65–80 (2006)
15. Fahland, D., Gierds, C.: Using Petri nets for modeling Enterprise Integration Patterns. Tech. rep., bpmcenter.org (2012), `http://bpmcenter.org/wp-content/uploads/reports/2012/BPM-12-18.pdf`
16. Frantz, R.Z., Quintero, A.M.R., Corchuelo, R.: A domain-specific language to design enterprise application integration solutions. Int. J. Cooperative Inf. Syst. 20(2), 143–176 (2011)
17. Friesen, A., Theilmann, W., Heller, M., Lemcke, J., Momm, C.: On some challenges in business systems management and engineering for the networked enterprise of the future. In: Ardagna, C., Damiani, E., Maciaszek, L., Missikoff, M., Parkin, M. (eds.) Business System Management and Engineering, Lecture Notes in Computer Science, vol. 7350, pp. 1–15. Springer Berlin Heidelberg (2012)
18. Gardey, G., Lime, D., Magnin, M., Roux, O.H.: Romeo: A tool for analyzing Time Petri Nets. In: CAV. pp. 418–423 (2005)
19. Gierds, C., Mooij, A.J., Wolf, K.: Reducing adapter synthesis to controller synthesis. IEEE T. Services Computing 5(1), 72–85 (2012)
20. Google: Checkout, `https://checkout.google.com/`, [retrieved Oct 19, 2012]
21. Grahlmann, B., Best, E.: PEP - more than a petri net tool. In: TACAS. pp. 397–401 (1996)
22. He, W., Xu, L.D.: Integration of distributed enterprise applications: A survey. IEEE Transactions on Industrial Informatics PP, 1–9 (2012)

23. Hentrich, C., Zdun, U.: A pattern language for process execution and integration design in service-oriented architectures. T. Pattern Languages of Programming 1, 136–191 (2009)
24. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
25. Ibsen, C., Anstey, J.: Camel in Action. Manning Publications (2010)
26. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
27. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. STTT 9(3-4), 213–254 (2007)
28. Konda, M.: Just Spring Integration - Enterprise Application Patterns and Messaging. O'Reilly (2012)
29. Kristensen, L.M.: A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future. In: Petri Nets 2010. LNCS, vol. 6128, pp. 39–42. Springer (2010)
30. Kristensen, L.M., Westergaard, M.: Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In: FMICS'10. LNCS, vol. 6371, pp. 215–230. Springer (2010)
31. Kwantes, P.M.: Design of clearing and settlement operations: A case study in business process modelling and analysis with petri nets. In: Jensen, K. (ed.) Proceedings of the Seventh Worskhop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. pp. 217–236 (Oct 2006)
32. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer Performance Evaluation / TOOLS. pp. 200–204 (2002)
33. Liu, D., Wang, J., Chan, S.C.F., Sun, J., Zhang, L.: Modeling workflow processes with Colored Petri Nets. Comput. Ind. 49(3), 267–281 (Dec 2002)
34. Lytra, I., Sobernig, S., Zdun, U.: Architectural decision making for service-based platform integration: A qualitative multi-method study. In: WICSA/ECSA. pp. 111–120 (2012)
35. Mederly, P., Lekavý, M., Závodský, M., Návrat, P.: Construction of messaging-based enterprise integration solutions using AI planning. In: Szmuc, T., Szpyrka, M., Zendulka, J. (eds.) Advances in Software Engineering Techniques, Lecture Notes in Computer Science, vol. 7054, pp. 16–29. Springer Berlin Heidelberg (2012)
36. Mendes, J.M., Leitão, P., Colombo, A.W., Restivo, F.: High-level petri nets for the process description and control in service-oriented manufacturing systems. International Journal of Production Research 50(6), 1650–1665 (2012)
37. Mortensen, K.H.: Automatic code generation method based on coloured petri net models applied on an access control system. In: ICATPN. pp. 367–386 (2000)
38. Mosawi, A.A., Zhao, L., Macaulay, L.A.: A model driven architecture for enterprise application integration. In: HICSS (2006)
39. Panetto, H., Jardim-Gonçalves, R., Molina, A.: Enterprise integration and networking: Theory and practice. Annual Reviews in Control 36(2), 284–290 (2012)
40. Rozinat, A., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M., Fidge, C.J.: Workflow simulation for operational decision support using design, historic and state information. In: BPM. pp. 196–211 (2008)
41. Scheibler, T., Leymann, F.: A framework for executable enterprise application integration patterns. In: Mertins, K., Ruggaber, R., Popplewell, K., Xu, X. (eds.) Enterprise Interoperability III, pp. 485–497. Springer London (2008)
42. Schmidt, K.: LoLA: A low level analyser. In: ICATPN. pp. 465–474 (2000)
43. University of Aarhus - Department of Computer Science: CPnets - industrial use, `http://cs.au.dk/cpnets/industrial-use/`, [retrieved Nov 28, 2012]
44. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2), 292–333 (1997)