Petri Net Sagas

Jan Henrik Röwekamp, Manuela Buchholz, Daniel Moldt

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Abstract. This paper introduces an integration of the Saga pattern for eventual data consistency in microservice deployments with concurrency supplied by labeled P/T workflow nets. Main benefits are improved speeds due to parallelization and providing methods of verification on successful runs due to the well-studied underlying formalism of Petri nets. The Saga pattern offers an availability-oriented alternative to classic distributed transaction and locking mechanisms. Use cases are large systems, that use the shared-nothing data model, which is typically fulfilled by microservice applications. Sagas are traditionally defined by a sequence of local transactions, but often concurrency between transactions is possible. This paper presents a solution to model concurrent Sagas with P/T workflow nets, called Petri net Sagas. As a proof-of-concept an implementation shows, how to translate Petri net Sagas to reference nets and to execute them using the Renew simulation core. The implementation is available as a library for the Java programming language.

Keywords: Data consistency, Microservices, P/T nets, High-level Petri nets, Workflow nets, Saga pattern, Shared-nothing data model

1 Introduction

The development of large-scale systems went through large changes of paradigms throughout the last decade. From the deployment of large monolithic applications on a bare-metal server the architectural design of applications was heavily influenced by cloud technology, which was commercialized in the 2010s. One of these architectural design shifts introduced the heavy use of the microservice pattern. For an extended introduction see e.g. [21]. It evolved from "service-oriented architectures" (SOA) and is rooted in web technologies.

Microservices decompose an application into small and independent parts, which are usually managed by separate development teams each. A lot of benefits can be reaped by utilizing microservices, e.g. scaling up the overall development workforce more easily. With the higher flexibility, new drawbacks are

introduced though. Microservice architectures usually use a shared-nothing data model, meaning that every service features its own data model and database and redundancy of data is inevitable.

Microservice applications may span over hundreds or more individual services. Therefore traditional database consistency technologies like distributed commits, that utilize locking and focus on providing consistency over availability, might not be feasible in this architecture anymore. This is due to frequent and complex queries and an increased chance of node failure due to the higher count of components in the system. According to the CAP-theorem [6] both, consistency and availability cannot be provided at the same time, when the network encounters partitioning or - more generally speaking - experiences node or link failures. Being far ahead of its time back then, the Saga pattern was introduced in the late 80s [12]. It is a protocol designed to be used instead of distributed transactions and features availability over consistency and promotes eventual consistency. Multiple local transactions are incorporated into a large so-called Saga spanning multiple services, hence the name. It ensures atomicity, consistency, and durability (ACD) but leaves the isolation up to the developer and the application level. With the rise of microservices, the Saga pattern found its revival rather recently [22].

The Saga pattern is usually implemented using state machines in mind, defining a process step by step. Possible concurrency is not utilized and therefore the approach misses out on the potential for speed-ups. Only very few approaches exist, that apply parallelism to the Saga pattern, like e.g. [18] in the context of and limited to the process calculus SOCK. Therefore the presented paper provides a solid base for concurrency in Saga pattern applications in the shape of Petri nets: Petri net Sagas. A general discussion on the approach is given as well as the description of a proof-of-concept implementation of the results.

Petri net Sagas are simple (labeled) P/T (workflow-)nets, that need to be provided by the developer specifying the Saga. In our underlying proof-of-concept implementation, the Petri net Saga is translated into a reference net [15], a high-level Petri net formalism with similarities to object-oriented programming. They then can be run by the Renew simulator [16] inside a library transparent to the developer.

Our main contributions within this paper are the easy to use definition of Petri net Sagas along with the translation mechanics to Reference nets for failure aware, concurrent, and transparent execution of Petri net Sagas. We thereby aid the development of Saga based consistency implementations by providing a mean to use a graphical representation of a Saga in the shape of a workflow net. The method also alleviates the complexity of implementing concurrent steps by hand within a Saga, especially in the Java programming language, as our proof-of-concept implementation is also Java-based. Java is one of the main programming languages in microservice environments after web technology languages like JavaScript[1].

[1] see e.g. https://tsh.io/state-of-microservices/#programming-languages

1.1 Motivation and Goals

Integrating services and providing eventual data consistency across multiple heterogeneous services can be a daunting task to achieve. Introducing additional concurrency usually increases the complexity even further. Therefore relying on a well-researched Petri Net formalism can help to design efficient and responsive microservice applications.

While simple Sagas can be described easier with less complex formalisms, Petri nets offer the possibility to describe more sophisticated procedures and by that are more future proof for further research. An additional motivation is the possibility to use model checking methods on Petri nets. Verification is only sparsely discussed in the context of the Saga pattern on some websites and online forums. There are to our best knowledge no citable sources related to Saga pattern verification.

The data, that is required by different services and might be required to be passed along is also not explicitly addressed by the Saga pattern. Using Petri nets as a base enables the notion of places and markings to model this kind of relations.

In summary, the goals of this paper are to provide a Petri net-based extension to the Saga pattern. This is done to enable concurrency and data allocation directly in the persistence model as well as providing ways to run formal verifications on the persistence operation more easily.

1.2 Example

To follow along the ideas presented in this contribution an example will be helpful. The example will focus on an large online book store, that is realized as a microservice application. Seven departments are considered: Order management, an inventory control system, pricing, a payment provider, accounting, fulfillment and finally customer notifications. Each department features an own system, with own data models and databases. The processing of purchases and by that interaction of the department systems is orchestrated by a classic sequential Saga, that will be described later in more detail.

2 Basics

The required basics for the addressed context are described briefly. The core aspects in the shape of microservices and the Saga pattern itself are introduced. This section also contains a basic introduction to reference nets, as they are a less generally known formalism of high-level Petri nets and differ in quite some aspects from other formalisms.

2.1 Microservices

The term *microservice* is not globally uniquely defined. As it, therefore, leaves room for interpretation, this section explains microservices as they are understood in the context of this paper. A microservice architecture consists of loosely

coupled components and follows the Unix philosophy to "make each program do one thing well". They require additional infrastructure components and severely increase the overall complexity of the system. Individual microservices are small[2] and rather easy to maintain and are usually tied to one specific development team. However, microservices are no silver bullet and their use should always be considered thoroughly beforehand.

Microservices use a shared-nothing data model, meaning every microservice manages its very own database. Multiple data models, as well as database technologies, can coexist in microservice deployments. Despite this freedom, eventual consistency must be guaranteed for the overall system to achieve meaningful results. Technologies like e.g. X/OPEN DTP (e.g. [13]) or other distributed transaction and locking mechanisms can slow down the system substantially as well as introduce deadlocks. The increased number of associated nodes in a microservice deployment increases the overall risk of failures in the system. Using locking mechanisms spanning multiple nodes may induce large downtimes upon failure of a single node.

This is usually tackled by relying on availability-oriented methods like the here discussed Saga pattern for data consistency.

2.2 Sagas

A Saga [12] is a set of node-local transactions alongside an order in which to invoke them. A Saga also requires the transactions to be of one of three specific types: Repeatable transactions, reversible transactions, and one so-called *pivot*-transaction. All reversible transactions need to precede the pivot-transaction, while all repeatable ones need to follow the pivot-transaction. The pivot-transaction may either succeed or fail and decides, whether the reversible transactions need to be reversed or the repeatable transactions are called. Sagas are thereby always *cut into three distinct* parts consisting only of the aforementioned transaction types each.

Example: Online book store For a more real-world view the earlier introduced example is considered. The scenario involves a customer, who orders a book and pays using a credit card. When the customer clicks an "order now" button in the interface, first an order id is generated and saved by the order service (first transaction). The book then needs to be reserved in the system managing available supplies with the order id (second transaction) and the current price needs to be fetched for payment processing (third transaction). After that, the credit card needs to be charged the amount required to pay for the book and shipping by an external payment provider (fourth transaction). After successful payment, the order gets transferred to the accounting system to be archived (fifth transaction) and to a fulfillment service to enqueue the shipping

[2] There is no explicit size requirement. However, due to the limit to one functionality, microservices tend to be much smaller in size than multi-purpose or monolithic applications.

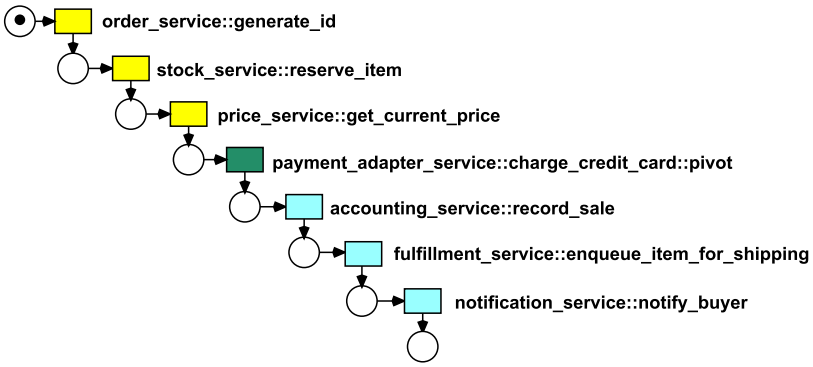

Fig. 1. Sequential (classic) Saga displayed as a labelled P/T net.

process (sixth transaction). Finally, the user receives a receipt via e-mail by the notification service (seventh transaction).

A Saga defines the steps from the first to the seventh transaction. Observe, that transactions one, two, and three are all reversible, as an order id may be deleted again and an item reservation may be canceled to make it available again. Reversing a credit card charge by the external payment provider is not that easy, hence it is considered the pivot-transaction in this case. It decides, whether the whole Saga fails or succeeds. Transaction five to seven are all repeatable and have no non-technical reasons to fail. Upon technical failures, e.g. if the respective service is unresponsive, it will be repeated until it succeeds. The whole example is also shown as a linear P/T net in figure 1. Note, that the syntax for describing the services and commands, separated with a double colon ("::"), is not specific to general Sagas, but used later in the here contributed Petri net Sagas and is already used here as well for consistency reasons.

Even in this small example, possible concurrency can easily be spotted. For example (if this is within the corporate guidelines) the user might get a receipt e-mail independent of the enqueueing of the shipping process or accounting and fulfillment might be notified at the same time. Also, the current price can be fetched regardless of the progress in generating an order ID and reserving the item, to name some possibilities. This example will be shown again later within the paper with applied concurrency.

Execution of a Saga There are two major approaches on how to implement the execution of Sagas. The related literature (e.g. [22]) describes them as "orchestration" and "choreography", which are terms also used for similar concepts in other microservice related research subjects.

Choreography is a classic fully decentralized approach, where each service only knows whom to contact next to proceed with a specific Saga. Because

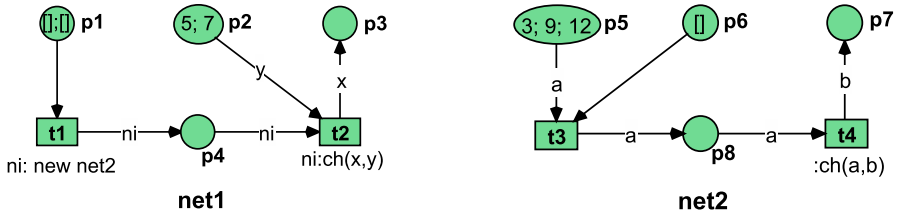

Fig. 2. A reference net example consisting of nets *net1* and *net2*.

services proactively call subsequent services the overall system is coupled more tightly. This also complicates changes to the Saga and deployments.

Orchestration in contrast relies on a centralized coordinator, that holds all knowledge over the entire Saga process and the network coordinates of all involved services. Maintainability and more loose coupling (every service is only called within a Saga but never calls) are bought by introducing a single point of failure into the system. Despite being the "impure" alternative, the orchestration approach is often the preferred one in real-world applications. The reasons are, that downtimes are more uncommon than the aforementioned deployment problems and modern infrastructure solutions may provide backup solutions in case of a node failure.

2.3 Reference Nets

The prototype implementation of the presented contribution uses the reference net formalism [15] in its underlying implementation. Therefore a quick introduction to the formalism is given.

Reference nets incorporate the "Nets within Nets"/"Object Nets" approach [29] with graph transformation and synchronous channels [8]. They feature the concept of *net instances*, which can be instantiated from nets like objects can be instantiated from classes in object-oriented programming. While their formal definition [15] is language-agnostic, they are almost always considered in the context of the Java programming language, as the simulator Renew [16] implements them. Tokens are references to data structures or other net instances. Reference nets are therefore well suited to construct complex hierarchical models. However, in the context of Petri net Sagas, they are chosen to support execution due to their seamless integration into object-oriented programming and their ability to directly handle function return values. Reference nets can use synchronous channels for multidirectional information passing to and from other net instances referenced by tokens.

In figure 2 a simple example of a reference net is given to illustrate the concept. The syntax is the Java-based one used by the Renew simulator.

$t1$ may fire twice, generating a new net instance of *net2* each time. $t2$ then may fire synchronously with $t4$ swapping one token each between the net instance

net1 and one of the net instances of *net2*. After all firings, the net locks with two tokens each of color 3, 9, or 12 in place $p3$ and one token of color 5 in one net instance's place $p7$ and of color 7 in the other net instance's place $p7$.

3 Related Work

Several related fields of research with respective publications exist. To our best knowledge, there is no direct approach to support eventual data consistency with concurrent operations across micro or web services using Petri nets as base formalism. A direct overview over Saga implementations is given by [28], which will be addressed again in section 5.5.

There, however, have been some considerations regarding the implementation of parallelized Sagas. They are related to process calculi, like [18], which considers implementing parallel Sagas in the context of the process calculus SOCK.

Some research integrating services with Petri nets or using them in other service-related work have been presented. [4] considers performance on web services, that are modeled with Petri nets but the focus lies on the mathematical model behind and not on data consistency questions. Similarly, [17] considers web service performance and elasticity of resources but focuses on business processes and service allocation, instead of inter-service optimization. Other works use specific Petri net subformalisms, like stochastic nets (e.g. [5]) or timed nets (e.g. [10]), to model, analyze, or interact with service structures.

[19] considers service choreography using the approach of Open Nets. The work addresses formal realizability aspects of certain message structures. While the topic is similar to our contribution, Petri net Sagas in general do not address the actual construction of orchestrations or choreographies, as they are left to the modeler. In this regard, the work of [19] can be used as a base to aid the construction of choreographies for Petri net Sagas.

Work related to running Petri nets in the cloud or Petri net-based cloud and web- and microservice technologies has also been presented. [26] and [7] considered containerization with Petri net execution and [25] contributed a framework to control the scale of a distributed Petri net simulation from within the simulation itself. [23] investigated the integration of web service technology with Petri nets but did not emphasize consistency.

From a consistency point of view, most research does not incorporate the shared-nothing data model. A respective example is the research area concerning sharded or replicated databases.

4 Integrating Sagas with a Petri Net Formalism

To incorporate Sagas with a Petri net-based formalism multiple aspects need to be considered. As the intended user group extends to developers, who are not experts in the field of Petri nets, the formalism needs to be kept simple to remain appealing. Also, as verification possibility is one of the claims, using a

formalism with a thorough verification tool support is desirable. Further, being a graphical representation of a process a clean representation of the Saga, that minimizes boilerplate elements, is desired.

While choreography based Sagas may seem more appealing from a scalable system point of view, it has already been motivated in section 2.2, that most real-world applications use orchestration-based Sagas instead. To address this and also because running a net simulation on a centralized node avoids distributed net simulation issues, in this paper Petri net Sagas are only presented with the orchestration application type in mind. Possible extension to choreography based applications is discussed in the outlook in section 8.

4.1 Base Formalism

As the example in figure 1 shows, simple P/T nets may be sufficient to model a Saga. To address the specific service and desired endpoint a label should be added to each transition. Using P/T nets yields the direct benefit of using the most widely known kind of Petri net formalism. Developers, who are unfamiliar with the concept of Petri nets, can rely on various courses and literature to acquire the basics required. Also, there are numerous P/T net editor tools to generate nets with. P/T nets are not Turing-complete and being a simple vector addition system gives lots of mathematical possibilities. Verification on P/T nets is well studied and established tools are available. Despite being well suited for successful Petri net Sagas, P/T nets have insufficient means to model different results from a single transition firing. This drawback, however, is not problematic, as will be addressed in more detail in section 4.2.

4.2 Failed (Pivot) Transactions vs. Transitions

The first problem to address is the possibility for transactions to fail. It is important at this point to differentiate between transactions and the transitions displaying them.

Methods to make a transition fail do exist, putting its consumed tokens back into its preset places. This behavior was introduced by [1] and [14] in the shape of task-transitions. However, transitions in Petri net Sagas are supposed to model transaction invocations, and even upon a failed transaction, its invocation in fact was successful. Also letting a transition fail (by using a task-transition) would not yield a possibility to handle these kinds of failure, as the marking before even firing the transition and after failing would be identical (except for other concurrent actions concerning the specific locality during the failed firing). In a Saga, a failure of a single transaction influences further processing of other transitions/transactions, and by that, it is not limited to its locality. Therefore specific handling of failed transactions is required, which introduces additional difficulties using the P/T net approach. Adding separate transitions for failed and successful transactions runs would require a change in P/T net semantics to run these directly (as the transition to be fired would not be chosen by activation only anymore), which we decided is not desirable.

Another problem occurs, when one concurrent branch of the Saga fails early and synchronization in the Saga later on may never happen. To illustrate this, consider the (sequential) Saga from the example in figure 1 alongside its discussed ideas for concurrency. The *stock_service* will require the order id generated by the *order_service*. However, the *price_service* is independent of any order id and thus may fire concurrently with both the *stock_service* and the *order_service* transactions. If for some reason, the *order_service* fails and gets rolled back, the *stock_service* transaction will never be invoked. Still, the result of the *price_service* awaits the result of the *stock_service* to join and proceed with invoking the *payment_adapter_service* pivot-transaction, which will never occur. Therefore some kind of global failure handling needs to be implemented into each concurrent transaction and their results. A process, that is deterministically computable and error-prone if done by hand and therefore undesirable to do directly in the P/T net.

Failures of compensatable transactions are sparsely discussed in literature. As an additional feature we therefore decided, that pre-pivot-transitions of Petri net Sagas are also allowed to fail and rollback the Saga, giving the developer more freedom in implementations. It is, however, recommended to model transactions, that have an actual chance to fail, that is non-negligible, as pivot-transactions.

As failure handling transitions essentially do not add to the expressiveness of the model, we decided to let Petri net Sagas only explicitly model the successful cases and deduce the failure handling from these during execution. The model including the failure paths is still interesting though when considering verification purposes of the net. More thoughts on this are discussed in section 5.8 in the context of our implementation.

As Sagas describe a course of actions to pursue a specific goal, modeling them with workflow nets [2] is a natural fit and hence they are the type of nets we recommended to model Petri net Sagas.

4.3 Passing Data

Executing (local) transactions during any step of a Saga usually requires some kind of data from previous transaction(s), or else they could be run concurrently. We assume the concurrent execution of different transactions implies no data dependencies exist between the transactions. Therefore a transaction is only dependent on the data of transactions, that are invoked before itself.

Also as individual microservices implement their functionality in their own domain, large exchanges of data between microservices can be assumed to be uncommon. Therefore it is sufficient to provide each transaction invocation with a set of results that occurred before its invocation in the current Saga.

Possible additions may explicitly state required data but for now, we will use this rather simple approach.

As the specific required data can be deduced from the structure of the Petri net Saga it is not necessary to explicitly model the data. It is already expressed by the command flow black tokens moving through the Petri net Saga during execution.

4.4 Syntax

Overall a P/T net representation of a successful Saga run was chosen to model the Saga because it is sufficient to model every aspect of the Saga, while the remaining aspects can be deduced implicitly. Labels hold information over the addressed service and the service endpoint (command) delimited by a double colon ("::"). The pivot-transaction is also specifically labeled with the "::pivot" suffix. The P/T net cannot be simulated directly, due to possibly failing transactions discussed earlier and therefore direct verification on the P/T net only holds for cases of successful runs. However, depending on the implementation, additional verification means are available. Section 5.8 addresses this topic in the context of the here presented implementation.

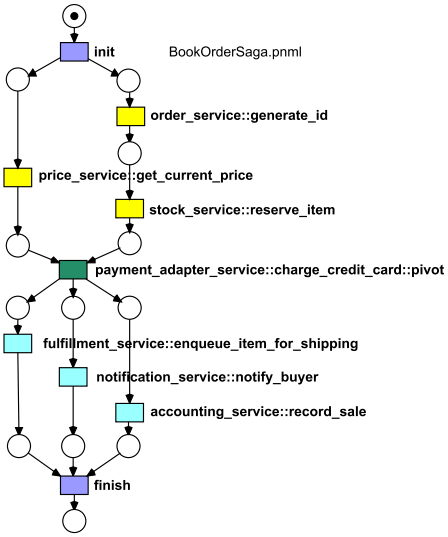

Fig. 3. Petri net Saga displayed as a labelled P/T net.

Petri net Sagas also feature specific "init" and "finish" transitions to enable the implementation to add initialization and cleanup steps around the execution of the Saga.

An example Petri net Saga (workflow net) of the concurrent version of the sequential Saga presented in section 2.2 is shown in figure 3. The colors were chosen to represent the different transaction types as well as clarity on the relocation of transitions and arcs from the linear example. However, using colors is not mandatory, as they do not bear any semantic meaning for the Petri net Saga definition itself.

5 Implementing a Proof-of-concept Prototype

To illustrate the concept of Petri net Sagas, the general approach to implement an execution environment for them will be presented. The section is oriented towards our proof-of-concept implementation. Figure 4 shows the general approach on how Petri net Sagas are processed by our implementation.

5.1 Basic Approach

We decided to execute the Saga by translating the P/T net representation into a higher Petri net formalism first: Reference nets. We did so to enable the possibility to be able to also supply more complicated Saga definitions using reference

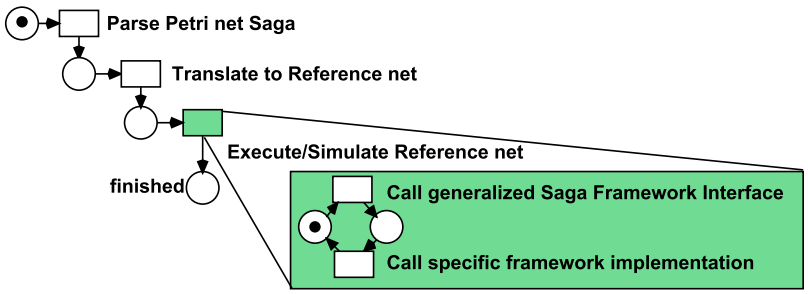

Fig. 4. Global view on the process implemented in the Petri net Saga interpreter.

nets. With reference nets it is also possible to easily model transaction failures while staying inside the reference net semantics. Further the powerful simulator Renew based on the Java programming language is available to directly execute reference nets as code. Finally, verification tasks can still be done to some degree on the more complex reference net variant, due to recent publications, like [31].

Using reference nets, however, introduces the requirement to transform P/T net Saga definitions into their reference net counterparts. During the translation, additional places and transitions get added to express the failure, repeatable, and rollback behavior to the net model. This is crucial for the advanced verification thoughts presented at the end of the section.

After being translated to a reference net representation the net will be simulated by an embedded Renew simulator. During the simulation, the net will promote calls to an abstract interface for Saga framework interaction. As different implementations of the Saga pattern exist, this is held as general as possible. Behind the general interface, a specific implementation handles the calls and passes them to the specific framework it is written for. Upon reaching the marking with just one token in the finishing place (in the workflow net sense) or the "rolled back" place, which will be introduced later in this section, the simulation terminates.

The basic syntax of the Petri net Saga is identical to the one presented in figure 3 and discussed in section 4.4.

5.2 Data Format and Parsing

The Petri net Saga is expected to be supplied in the PNML[3] format. To construct the specific inscriptions, transitions, and places required to model the reference net version of the Petri net Saga, a specific data format was constructed. It is based on a simple net data structure featuring net, place, transition, and arc objects and adds Saga specific content like pivot-transitions and mirror transitions (for reversible pre-pivot-transactions).

[3] Petri Net Markup Language - http://www.pnml.org/

The data is then transformed into its reference net counterpart, which is discussed in detail in the next section. To be fed into the Renew simulation core the data is then transferred back into a reference net specific variant of PNML. The reference net representation is not intended to be shown to the developer by default. However, it can be saved during the process to aid in troubleshooting and debugging scenarios. It is also possible to directly supply advanced reference net-based definitions and skip the transformation part.

5.3 Transformation to reference nets

Reference net transformation tools like the *Renew Metamodelling and Transformation* approach [20] exist, that might help to transform the Petri net Saga to its reference net counterpart. The transformation of the net requires some steps, which are not limited to the locality of a net element, e.g. differentiating between transitions before and after the pivot-transition. Therefore existing metamodelling tools could unfortunately not be used, as they focus on locality-based transformation and cannot handle global constraints.

The transformation process itself consists of five parts: creation of the transitions for compensating transactions, transformation of the labels, addition of guards and data passing and finally detecting failures. To be easier to follow, it will be described alongside the book store example. Figure 5 shows the finished and translated reference net from figure 3. The net is much more complex and displays detailed processing.

The procedure for the example starts with the workflow net representation depicted in figure 3. First the entire net structure prior to the pivot transition is copied and its arcs are reversed. The result can be observed in the upper right part of figure 5. This leads to an additional net part, that is able to execute the compensatable transactions in reverse order (or concurrently if no order was defined by the original Saga net).

In the lower part of the net for repeatable transactions (depicted as light blue transitions) additional red repetition transitions are added to retry transactions after failures.

After that all the labels are converted into Renew-specific calls: "step" calls in the original part of the net, and "rollback" calls in the newly copied part from the last step. These can be seen in the transaction inscriptions. During the transformation of the labels data passing ("result" inscriptions on the arcs) and guards are added, that handle possible failures and decide which route to take throughout the net. An example for such a translation is:

```
                price_service::get_current_price →
action result2 = Saga.step("price_service", "get_current_price",
                           [result1]);
```

Once data handling and guarding is finished, global running and failure indicators can be added. These are shown as red and green places and transitions in the upper left part of the figure. The "running" and failed place use so-called

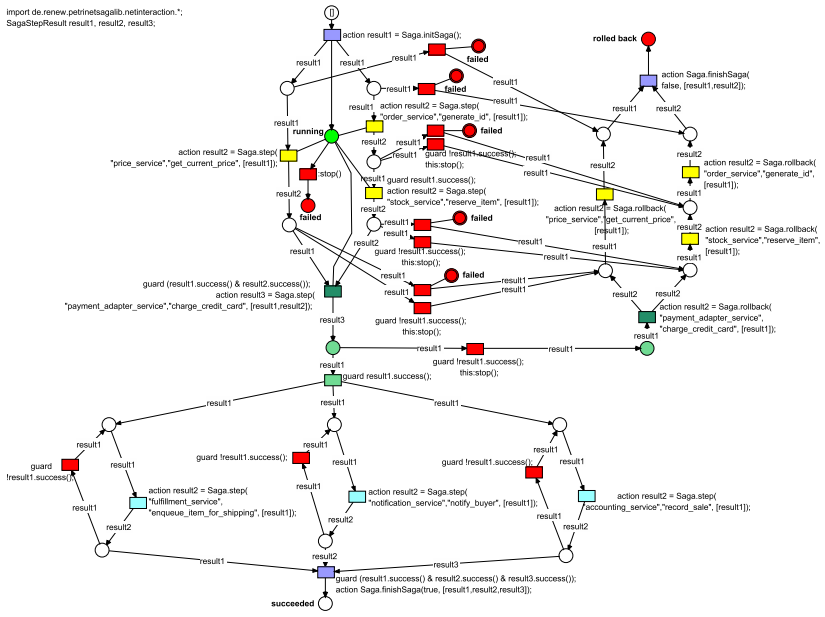

Fig. 5. Reference net generated for the example Petri net Saga in background.

test arcs (without a tip), that are read-only, non-consuming and non-blocking for concurrent operations. Alongside these indicators, red failure handler transitions are added, that can be observed in the figure as vertical array of red transitions between the two groups of yellow transitions.

Note, that the red "failed" places with double outline are so called virtual places and have by definition the same identity as the "failed" place with just one outline. Virtual places are used here to keep the net less cluttered but bear no semantic meaning. There are always two possibilities for failures: one, if the last transaction prior to this one failed, and one, if another transaction on a different concurrent branch failed. Synchronous channels are used for tidiness here ("`this:stop()`").

5.4 Execution Engine

Due to their nature reference nets can be executed directly by the simulator. The execution engine is given in our implementation by using the simulation core of the Renew simulator. It is embedded into the Petri net Saga library and invisible to the programmer and administrator, except for its log outputs and attached libraries. For further information in regards to the usage of the simulator please refer to the Renew user guide [16].

5.5 Generalized Framework Interface

Different available implementations of the Saga pattern exist. They are usually supplied in the shape of a framework and differ in functionality and licensing. An implementation specific to one framework is possible but limiting. Therefore a general-purpose interface is put in front of the implementation for a specific framework. The actually used framework can thus be swapped when developing with Petri net Sagas.

Available implementations are e.g. the Eventuate framework[4] and the Axon framework[5], which both are commercial. Micro-profile-lra[6] is an open source implementation and is based on the Narayana[7] project. A survey on these frameworks can be found in [28].

The implementation expects the usage of a supplied class to pass data from one Saga step to the next. The interface can then be extended for a specific framework by implementing the behavior before the actual invocation of the Petri net Saga, after its completion, and the callbacks from the actual step or rollback actions.

5.6 Deployment and scope

The intended use of our Petri net Saga implementation is as a Java library in the development of microservice applications. Note, that enforcing consistency using the Saga pattern is just one aspect of a microservice deployment. Therefore, if considered in a larger scope, this work only addresses the orchestration of Sagas being done based on Petri nets. Other components like commands, services and service descriptors, message systems, etc., that are not Saga-related, need to be implemented and supplied separately to realize an entire microservice deployment.

The participants of a specific Saga do not need to be aware of the usage of Petri net Sagas instead of conventional ones. This is due to the fact, that participants in orchestration-based Sagas - as described at the beginning of this contribution - are always only contacted by the orchestrator and do not possess information about the global process. However, they still need to implement the specific command method or class used by the respective Saga framework.

5.7 Implementation using the Eventuate Framework

Being one of the available frameworks for handling Saga patterns, the Eventuate framework is one of the most advanced ones and most dedicated to the Saga use case. Despite using an *All rights reserved* license, it follows the visible source[8]

[4] https://eventuate.io/ - All URLs have been checked in January 2021
[5] https://axoniq.io/
[6] https://github.com/eclipse/microprofile-lra
[7] https://github.com/jbosstm/narayana/tree/master/rts/lra
[8] https://github.com/eventuate-tram/eventuate-tram-sagas

approach. For the aforementioned reasons, we implemented a sample adapter between our general-purpose interface and the Eventuate framework.

Evenutate is tightly coupled to microservice frameworks such as the Spring Framework or Micronaut. It also relies (as one possibility) on Apache Kafka[9], which again relies on Apache Zookeeper[10]. Eventuate also utilizes a local backing database for the orchestrator. Globally seen, this setup is well known in microservice environments.

Despite its restrictive licensing, the code is designed to be very open for additions, usually using interfaces instead of direct references. The context of dependency injection[11] usually adopted in microservice environments supplies loose coupling and eases integration of the reference net adapter. For the adapter, a new implementation of the central management component has been introduced, which moves the execution to the reference net implementation instead of using a state machine.

5.8 Verification

Running verification tools and considerations on the Petri net Saga itself offers possibilities to express information about successful runs of the Petri net Saga. The translation to reference nets presented within this implementation, however, offers the possibility to discuss all possible runs. This is in most cases more desirable. Performing the transformation without restrictions on inscriptions and on the reference nets concepts changes the formalism to reference nets.

Reference nets are Turing-complete in their general form. However, there has been the interesting recent contribution MoMoC [31], that constructs specialized reachability graphs for restricted reference nets. Undergoing attempts at our research team try to incorporate these with the tool RenewKube [25] to scale up reachability computation and the first results look promising.

Another attempt could be made by changing the semantics of the P/T Petri net Saga in a way, that transition firing is controlled by the behavior of the system instead of arbitrary firings of activated transitions. This would yield the possibility to construct P/T nets similar to the presented reference net translation, that models failures in separate transitions. Depending on the system successfully or unsuccessfully completing a transaction the respective transition could be fired. In [1] the well-known concept of workflow transitions has been extended to a reference net implementation that could easily be adapted for this. Extensions in that direction were addressed in [3] and [30].

6 Example and Usage

Our implementation can be downloaded from the project's website[11]. This section will briefly describe the implementation of custom use cases, as well as the implementation of the example presented throughout this paper.

[9] https://kafka.apache.org/
[10] https://zookeeper.apache.org/
[11] https://paose.informatik.uni-hamburg.de/paose/wiki/PetriNetSagas

6.1 Custom Use Cases

First and foremost a PNML file describing the Petri net Saga needs to be supplied, like the one supplied by the example application. Additionally, as the implementation is based on Eventuate Tram Sagas, the respective Eventuate libraries, as well as the Petri net Saga libraries need to be supplied to the application. A Java Spring[12] and Gradle[13] based empty sample project containing all required dependencies can be found at the download location as well.

To use the framework several classes need to be implemented to specify the respective actions to be executed when invoking the commands of the Saga. A custom result type extending our base type needs to be created and command classes need to be registered with the command registry. This also involves defining a custom aggregation implementation to aggregate multiple results of previous concurrent invocations. Participants need to register themselves under the service descriptor names with our command handle registry. If the setup of remote participants is not intended the service descriptor *local* could be used, though - if used exclusively - it defies the benefits of using a Saga framework in the first place. For further assistance with the Eventuate part of the implementation, please refer to the repositories and tutorials under its aforementioned website.

Handling a request with a Petri net Saga usually consists of instantiating the initial result (data) object of the request and creating a Petri net Saga by invoking the reference net Saga instance factory.

6.2 Example: Book store

Finally, an example of how to use our implementation of Petri net Sagas is presented. To set up the example on a local machine as well, please refer to the setup instructions at the download location. To run the example of our Petri net Saga implementation a Linux environment is recommended, as well as Docker and the tool docker-compose, although these are not strictly necessary, as well as a Java JDK, which again is necessary.

We will reuse the example shown in figure 3 in section 4.3. The example is based - like the empty sample project - on the Spring framework and Gradle. It uses Apache Kafka, Apache Zookeeper, the Eventuate Tram CDC service, Zipkin[14] and Mysql as backing services, which can be launched conveniently using docker-compose. The example itself defines a total of seven custom command classes and a total of seven participating services. Each command just prints to the command line, that it has been invoked and then returns. The example defines a single Saga with reference to the PNML net file. Results are aggregated just by combining their success-states with an AND-operation.

Upon starting, the consecutive and - where applicable concurrent - invocation of the steps can be observed in the log output.

[12] https://spring.io/
[13] https://gradle.org/
[14] https://zipkin.io/

7 Evaluation and Discussion

Based on the experiences with the prototype implementation our first evaluation of the method can be discussed.

The approach integrated rather well with the Saga pattern, at least with the Eventuate implementation, though the central management core needed to be reimplemented accordingly. Sagas are already designed with concurrency between different Sagas and Saga instances in mind. However, feedback time for a user or process usually depends on the execution of one specific Saga related to the request, and not multiple concurrent Sagas. Other Sagas executed by the system remain hidden by the transparency of the system. In the book store example presented throughout the paper this results in just four sequential steps instead of the original seven. Being a method of introducing concurrency, the performance gain depends heavily on the possibility to parallelize the process.

Traditional Saga systems hold some unused potential, as the robustness towards concurrency in the overall system is present, but not used within an individual Saga. Introducing the possibility of concurrency into a model designed as a sequential system implies several aspects to consider. By backing the Saga with a formal Petri net model we were able to mitigate these troublesome details to a degree. This provides a solid base for describing concurrent Sagas and successfully realizes the addressed potential.

On the downside, by our choice to express Petri net Sagas with labeled P/T nets, the possibilities to model data passings are (in this first version) limited. They need to be covered by the Petri net Saga implementation and also the final implementation of the system using the Petri net Saga library. Also, adding another layer of dependencies to the equation of a microservice system further increased its complexity.

Overall, we see the concept and the implementation as very promising and already plan certain use cases with its implementation in the context of the Renew simulator itself.

8 Conclusion

A novel method of defining concurrent Sagas based on P/T workflow nets has been presented. The method is intended to be used in persistence operations in availability-oriented microservice deployments. We have motivated, that performance and response time for single Sagas can be improved by using the method and outlined the use of verification methods on the underlying formal Petri net model. In addition to the presented concept, a sample implementation based on translation to reference nets and execution using a Renew simulation core and the Eventuate Tram Sagas framework has been contributed.

Outlook

Further research may include support for choreography-based solutions. The main challenges will include failure detection in concurrent branches.

Another extension could model specific data portions required by each transaction invocation. Depending on the use case this might reduce the passed data between orchestrator and local transaction invocation. The restriction to P/T nets of manageable size might be difficult to hold under these circumstances and the use of colored nets (by folding) might become more suitable.

In further research, the here presented concepts can be applied to distributed synchronous channels of distributed reference net simulations. The synchronous channels may only hold at most one *action*[15] inscription to be compatible with the pivot-transaction model. This enables a much better scaling of distributed reference net simulations [25] and is an extension of previous consideration in this direction (see our contributions [23] and [24]).

References

1. van der Aalst, W., Moldt, D., Valk, R., Wienberg, F.: Enacting interorganizational workflows using nets in nets. In: Becker, J., Mühlen, M., Rosemann, M. (eds.) Proceedings of the 1999 Workflow Management Conference Workflow-based Applications, Münster, Nov. 9th 1999. pp. 117–136. Working Paper Series of the Department of Information Systems, University of Münster (1999), working Paper No. 70
2. Aalst, van der, W., Hee, van, K., Houben, G.: Modeling workflow management systems with high-level petri nets. In: Michelis, de, G., Ellis, C., Memmi, G. (eds.) Proceedings of the 2nd Workshop on Computer-Supported Cooperative Work, Petri Nets and related formalisms. pp. 31–50 (1994)
3. Bendoukha, S.: Multi-Agent Approach for Managing Workflows in an Inter-Cloud Environment. Dissertation, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg (2017), `http://ediss.sub.uni-hamburg.de/volltexte/2017/8241/`
4. Bernine, N., Nacer, H., Aïssani, D., Alla, H.: Towards a performance analysis of composite web services using petri nets. Int. J. Math. Oper. Res. **17**(4), 467–491 (2020). https://doi.org/10.1504/IJMOR.2020.110847
5. Bhandari, G.P., Gupta, R.: Fault diagnosis in service-oriented computing through partially observed stochastic petri nets. Service Oriented Computing Applications **14**(1), 35–47 (2020). https://doi.org/10.1007/s11761-019-00279-5
6. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. p. 7. PODC '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/343477.343502
7. Buchs, D., Klikovits, S., Linard, A., Mencattini, R., Racordon, D.: A model checker collection for the model checking contest using docker and machine learning. In: Khomenko, V., Roux, O.H. (eds.) Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings. LNCS, vol. 10877, pp. 385–395. Springer (2018). https://doi.org/10.1007/978-3-319-91268-4_21

[15] *action* inscriptions are usually used in commands causing side effects. Our Renew extension to use logical functional language inscriptions will address the avoidance of side effects of actions, see [27].

8. Christensen, S., Damgaard Hansen, N.: Coloured petri nets extended with channels for synchronous communication. In: Valette, R. (ed.) Application and Theory of Petri Nets 1994. pp. 159–178. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
9. Donatelli, S., Haar, S. (eds.): Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11522. Springer (2019). https://doi.org/10.1007/978-3-030-21571-2
10. Entezari-Maleki, R., Etesami, S.E., Ghorbani, N., Niaki, A.A., Sousa, L., Movaghar, A.: Modeling and evaluation of service composition in commercial multiclouds using timed colored petri nets. IEEE Trans. Syst. Man Cybern. Syst. **50**(3), 947–961 (2020). https://doi.org/10.1109/TSMC.2017.2768586
11. Fowler, M.: Inversion of control containers and the dependency injection pattern. Web (2004), `https://martinfowler.com/articles/injection.html`
12. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. pp. 249–259. SIGMOD '87, ACM, New York, NY, USA (1987). https://doi.org/10.1145/38713.38742
13. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (1992)
14. Jacob, T.: Implementierung einer sicheren und rollenbasierten Workflow"=Managementkomponente für ein Petrinetzwerkzeug. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg (2002)
15. Kummer, O.: Referenznetze. Logos Verlag (2002), `http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=eng&id=`
16. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L., Haustermann, M., Mosteller, D.: Renew – User Guide (Release 2.5.1). University of Hamburg, Faculty of Informatics, Theoretical Foundations Group (nov 2020), `http://www.renew.de/`
17. Lacheheub, M.N., Hameurlain, N., Maamri, R.: Resources consumption analysis of business process services in cloud computing using petri net. J. King Saud Univ. Comput. Inf. Sci. **32**(4), 408–418 (2020). https://doi.org/10.1016/j.jksuci.2019.08.005
18. Lanese, I., Zavattaro, G.: Programming sagas in SOCK. In: Hung, D.V., Krishnan, P. (eds.) Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009. pp. 189–198. IEEE Computer Society (2009). https://doi.org/10.1109/SEFM.2009.23
19. Lohmann, N., Wolf, K.: Decidability results for choreography realization. In: Kappel, G., Maamar, Z., Nezhad, H.R.M. (eds.) Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings. Lecture Notes in Computer Science, vol. 7084, pp. 92–107. Springer (2011). https://doi.org/10.1007/978-3-642-25535-9_7
20. Mosteller, D., Cabac, L., Haustermann, M.: Integrating Petri net semantics in a model-driven approach: The Renew meta-modeling and transformation framework. Transaction on Petri Nets and Other Models of Concurrency XI **11**, 92–113 (2016). https://doi.org/10.1007/978-3-662-53401-4_5, `http://dx.doi.org/10.1007/978-3-662-53401-4`
21. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, Inc. (2016)
22. Richardson, C.: Microservices Patterns: With Examples in Java. Manning Publications (2019)

23. Röwekamp, J.H.: Investigating the java spring framework to simulate reference nets with Renew. pp. 41–46. No. 2018-02 in Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg (2018), https://opus.bibliothek.uni-augsburg.de/opus4/41861
24. Röwekamp, J.H., Feldmann, M., Moldt, D., Simon, M.: Simulating Place / Transition Nets by a distributed, web based, stateless service. In: Moldt, D., Kindler, E., Wimmer, M. (eds.) Petri Nets and Software Engineering. International Workshop, PNSE'19, Aachen, Germany, June 24, 2019. Proceedings. CEUR Workshop Proceedings, vol. 2424, pp. 163–164. CEUR-WS.org (2019), http://CEUR-WS.org/Vol-2424
25. Röwekamp, J.H., Moldt, D.: RenewKube: Reference net simulation scaling with Renew and Kubernetes. In: Donatelli and Haar [9], pp. 69–79. https://doi.org/10.1007/978-3-030-21571-2_4
26. Röwekamp, J.H., Moldt, D., Feldmann, M.: Investigation of containerizing distributed Petri net simulations. In: Moldt, D., Kindler, E., Rölke, H. (eds.) Petri Nets and Software Engineering. International Workshop, PNSE'18, Bratislava, Slovakia, June 25-26, 2018. Proceedings. CEUR Workshop Proceedings, vol. 2138, pp. 133–142. CEUR-WS.org (2018), http://ceur-ws.org/Vol-2138/
27. Simon, M., Moldt, D., Schmitz, D., Haustermann, M.: Tools for Curry-Coloured Petri nets. In: Donatelli and Haar [9], pp. 101–110. https://doi.org/10.1007/978-3-030-21571-2_7
28. Stefanko, M., Chaloupka, O., Rossi, B.: The saga pattern in a reactive microservices environment. In: van Sinderen, M., Maciaszek, L.A. (eds.) Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019. pp. 483–490. SciTePress (2019). https://doi.org/10.5220/0007918704830490
29. Valk, R.: Petri nets as token objects - an introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal. pp. 1–25. No. 1420 in Lecture Notes in Computer Science, Springer-Verlag (1998). https://doi.org/10.1007/3-540-69108-1_1
30. Wagner, T.: Petri Net-based Combination and Integration of Agents and Workflows. Ph.D. thesis, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg (2018), http://ediss.sub.uni-hamburg.de/volltexte/2018/8995/
31. Willrodt, S., Moldt, D., Simon, M.: Modular model checking of reference nets: MoMoC. In: Köhler-Bußmeier, M., Kindler, E., Rölke, H. (eds.) Proceedings of the International Workshop on Petri Nets and Software Engineering co-located with 41st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2020), Paris, France, June 24, 2020 (due to COVID-19: virtual conference). CEUR Workshop Proceedings, vol. 2651, pp. 181–193. CEUR-WS.org (2020), http://ceur-ws.org/Vol-2651/paper12.pdf