Dromi: A Tool for Automatically Reporting the Impacts of Sagas implemented in Microservice Architectures on the Business Processes

Sandro Speth[1], Uwe Breitenbücher[2], Sarah Stieß[1], and Steffen Becker[1]

[1] Institute of Software Engineering, University of Stuttgart, Germany
[2] Institute of Architecture of Application Systems, University of Stuttgart, Germany
`[lastname]@informatik.uni-stuttgart.de`

Abstract. Distributed transactions that span multiple microservices are more and more realized using the Saga Pattern. However, in case the interaction between microservices fails due to an Service Level Objective (SLO) violation, e.g., insufficient availability, the executed business logic gets significantly impacted when a saga compensates already executed operations. Unfortunately, analyzing such impacts manually and reporting found issues is too slow for modern systems. Therefore, we present *Dromi*, a model-based tool that traces the impacts of SLO violations across a microservice architecture and, if a violation results in compensations caused by sagas, creates an issue report about the violation's location and resulting impacts on the business processes. The target audience of this demonstration includes architects and developers, who will be shown how such impacts are detected automatically by Dromi.

Keywords: Microservices · Impact Analysis · SLO · Business Processes

1 Introduction

In modern software architectures, often independent microservices work together to implement business processes. However, many processes require distributed business transactions that span multiple services which do not support atomic commit protocols. Therefore, business transactions in microservice architectures are often realized using the *Saga Pattern* [6]. A saga is a sequence of local transactions executed by different microservices. If a local transaction of a microservice fails, the saga executes a series of compensating transactions to undo the changes made by the preceding local transactions that were completed successfully.

However, local transactions executed by microservices can quickly fail due to SLO violations of other services, which promise the quality of a service, such as availability or maximum response time. For example, if a microservice executes a local transaction that depends on the invocation of another service that is many minutes not available, the transaction fails because the other service violates its SLO guaranteeing a maximum downtime of 10 seconds. Thus, such SLO violations quickly result in undoing several already completed local transactions of different microservices due to a saga that executes compensating transactions for them. Of course, this heavily impacts the business processes implemented by the microservices since the executed functional business logic gets changed due to the violation of a non-functional SLO. Unfortunately, if many business processes are implemented by a large microservice architecture consisting of many independent services, (i) detecting the impact of SLO violations on the business processes and (ii) reporting analyzed issues is a complex challenge and can hardly be done manually since these tasks are very time consuming and error propagations are hard to detect manually during runtime.

To tackle these issues, in this demonstration, we present *Dromi*, a model-based tool that traces the impacts of SLO violations across a microservice architecture and, if a violation results in compensations caused by sagas, creates an issue report about the violation's location and resulting impacts on the business processes. To enable this, we also introduce the *Dromi Modeling Language (DML)* in this paper, which combines models for microservice architectures, business processes, and sagas. Developers can link the elements of the models with each other to specify their dependencies, which are then used by Dromi to derive impact traces automatically and to generate issues in case an SLO violation has an impact on a business process. Thus, issues are reported automatically without human interaction once the DML model is created. We demonstrate how Dromi and the Dromi Modeling Language can be used in a video[3] and provide all implementations as open source code in GitHub.

2 Motivating Scenario

This section describes the motivating scenario used for explaining Dromi. In the scenario, we consider the T2-Project [9], a microservice architecture for a webshop that implements a business process to order teas. The architecture includes several services to realize an order: (1) the *cart service*, (2) an *orchestrator service*, (3) the *inventory service*, (4) the *order service*, and (5) the *payment service*, which invokes a (6) *credit institution service*. To maintain the consistency of an order, the inventory service, order service, and payment service participate in a saga in which the orchestrator manages to roll back orders in the event of a failure. However, although the saga is implemented, it is typically not directly observable because it is often not modeled in the architecture. In particular, the effects of a failure on the business process itself are not directly visible because the saga logic is often hidden directly in the microservices' code. For example, if the credit institution violates an SLO, e.g., because the service is currently unavailable or the request times out before a response is sent, it cannot be reached by the payment service. Therefore, the payment fails and triggers a saga compensation, which has side effects on the business process. However, these side effects are hidden in the code of the individual microservices that the orchestrator controls. Therefore, it is not apparent what impact a non-functional SLO violation has on the functional level of the business process.

[3] https://youtu.be/3E90neB-iUY

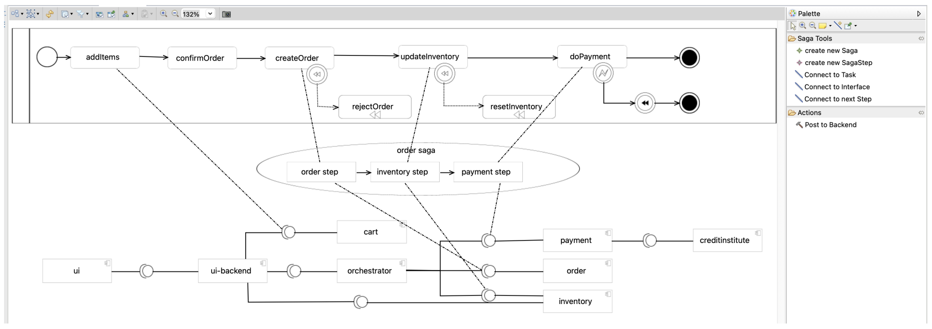

Fig. 1: Dromi Editor showing the motivating scenario modelled in DML.

3 The Dromi Modeling Language

The Dromi Modeling Language (DML) consists of three connected layers: (1) the *Architecture Layer*, (2) the *Business Process Layer*, and (3) the *Saga Layer*. Figure 1 shows a screenshot of our *Dromi Editor*, which is presented in detail in Section 4. We explain the layers based on this screenshot to give an overview.

The *Architecture Layer* describes the architecture of the system to be observed. Please note that DML is not bound to microservices but supports any type of component. Therefore, in DML, the Architecture Layer is realized as *UML Component Diagram*, where each component can provide and require multiple interfaces that other components can invoke. The Architecture Layer of the motivating scenario is shown on the bottom of Figure 1, which describes the microservices of the webshop, their interfaces, and their dependencies.

In the *Business Process Layer*, developers model the business processes implemented by the microservices described in the Architecture Layer. For example, Figure 1 shows on the top the order process implemented by the webshop microservices of the motivating scenario. Please note that these *business process models* are not executable but just describe the business logic that is implemented by the microservices. Thereby, a single activity contained in the process model can be implemented by one or more microservices. The Business Process Layer is realized using the business process modeling language *BPMN 2.0* since BPMN is widely used in practice. Another reason is that BPMN enables the modeling of *transactional sub-processes* and supports the concept of *compensation*, which is concerned with undoing activities that were already successfully completed because of their effects are no longer desired and need to be reversed. Thus, these BPMN modeling concepts can be used to describe the effects on the business process caused by microservice sagas in the case of compensation.

Business process models are often large. Thus, transactional behavior and compensation activities might be scattered throughout the model, which makes it messy to understand dependencies, e.g., in which situation which activities are compensated. Therefore, DML defines an optional *Saga Layer* located between the two other layers as shown in Figure 1, which is used for modeling sagas and their steps. Using the Saga Layer, business transactions can be separately modeled as sagas, independently of the Business Process Layer. For example, Figure 1 shows the *order saga*, which consists of an *order step* that maintains order information, an *inventory step* that removes ordered products from the inventory, and a *payment step* that executes the payment. The steps are executed in the described order.

To enable impact analysis across the layers, developers need to specify the dependencies of the elements in the three layers using *links*, which are shown as dashed lines in Figure 1. DML allows three ways to link layers and their elements: First, (i) a microservices interface can be directly linked with the BPMN activity realized by this interface. Second, (ii) if the microservice architecture implements one or more saga, links can be directed from a microservice interface to a saga step. Since each microservice interface can be involved in an arbitrary number of sagas, each interface can have multiple links to different saga steps. Finally, (iii) to relate saga steps to BPMN activities, links can be specified between them.

4 System Architecture of Dromi and Demonstration

This section presents Dromi, a tool that automatically detects and reports the impacts of sagas implemented in microservice architectures on the business processes they implement. Thereby, Dromi focuses on the impacts on processes caused by the compensation of sagas that result from SLOs violations of microservices, e.g., high latencies. The source code of Dromi is available on GitHub[4].

As depicted in Figure 2, Dromi consists of two parts: (i) *Dromi Frontend* and the (ii) *Dromi Backend*. The frontend is a graphical editor for DML models implemented as Eclipse EMF plugin. It enables importing BPMN 2.0 process models created with BPMN tools. Similarly, it enables importing architecture models created using the tool Gropius, which supports a UML Component Diagram-like notation [8]. Finally, the frontend enables drawing sagas in the middle layer and linking elements with each other as described in Section 3.

The Dromi Backend consumes DML models. For detecting SLOs violations, the tool *SoLOMON* [7] is integrated. SoLOMON automatically imports the architecture model from Gropius, with which developers map components deployed on Kubernetes to components of the architecture. Then, they model SLOs for the components via SoLOMON's frontend and send them to the SoLOMON backend, where the SLOs are transformed to PromQL queries and monitored through Prometheus. If an alert is triggered, i.e., an SLO is violated, SoLOMON creates an issue for the affected component in Gropius describing the violation. Furthermore, SoLOMON publishes an event that Dromi can subscribe for containing the violated SLO, affected component, time, and the issue ID.

When the Dromi backend receives this event, it executes the *Impact Analysis*, which starts in the Architecture Layer with the microservice that is affected by

[4] https://github.com/stiesssh/dromi-backend, https://github.com/stiesssh/dromi-models

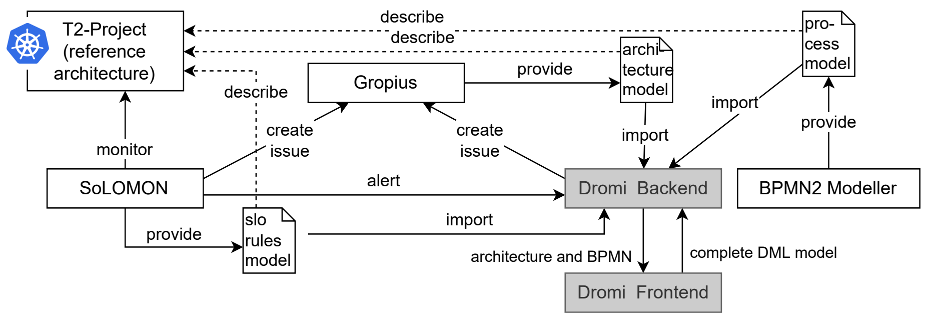

Fig. 2: System Architecture of Dromi.

the SLO violation. In this work, we assume that an SLO violation triggers a saga compensation. If a link exists from the interface of this service to a higher layer, the analysis follows the link until a BPMN task is reached. If no link to a saga or business process exists for the affected microservice, Dromi follows the call chain of the microservice architecture reversely to a service with a link to a higher layer that can be followed. The result of the impact analysis is an *impact trace* that includes the affected microservices, saga steps, and BPMN activities.

After analyzing impacts, the backend automatically executes *Impact Reporting* by creating an issue with the impact trace in Gropius [8], which is a *uniform issue management system* that integrates, e.g., GitHub and Jira. Thereby, it allows reporting issues independently of a microservice's actual issue tracker.

We recorded a video[5] to demonstrate the following example with Dromi: Assume the credit institute service in Figure 1 violates its availability SLO and is unavailable when an order should be checked out. Then the payment service cannot reach it to perform the payment. As a result, the orchestrator service rolls back the saga. Dromi analyses the impact of the SLO violation and reports an issue stating the violated SLO, time, *credit institute service* and *payment service*, the *payment step* of the saga, and the failed business activity, i.e., *doPayment*.

5 Related Work

Hanemann et al. [3] decide on system recovery actions on the cost caused by the SLO violation which caused the failure. They analyze the impact of the violation on the directly dependent services to estimate the cost. Unlike Dromi, they do not analyze the propagation of a violation along the call chain through the architecture. Furthermore, they do not consider impacts on the business process. Mohamed et al. [5] calculate architecture service routes, i.e., sequences of services connected through interfaces, and use them to consider failure propagation. However, they do not consider the process and lack the focus on SLOs, as they consider failures in general. Kleehaus et al. [4] developed the MICROLYZE

[5] https://youtu.be/3E90neB-iUY

framework that uses static and dynamic data for dynamic microservice architecture recovery. They also include the business process in their recovery and map tasks to interfaces. For future work, they mention a failure-impact visualization. However, they do not consider patterns. In the domains of requirements engineering and process alignment, authors connect elements from different modeling languages to ensure conformance between process and architecture model [1, 2]. However, they use direct connections, which are not sufficient to achieve our objectives, as we must also consider patterns, which none of those works does.

6 Conclusion & Future Work

We showed that Dromi and DML enable the automated impact analysis of a non-functional SLO violation on the functional logic of business processes when transactions span multiple microservices following the Saga pattern. Thus, Dromi is able to trace the violation's impact from the architecture to the business process. In future work, we plan to integrate more microservice patterns in Dromi.

References

1. Aversano, L., Grasso, C., Tortorella, M.: Managing the alignment between business processes and software systems. Information and Software Technology **72** (2016)
2. Elvesæter, B., Panfilenko, D., Jacobi, S., Hahn, C.: Aligning business and it models in service-oriented architectures using bpmn and soaml. In: Proceedings of the First International Workshop on Model-Driven Interoperability. p. 61–68. MDI '10, Association for Computing Machinery (2010)
3. Hanemann, A., Schmitz, D., Sailer, M.: A framework for failure impact analysis and recovery with respect to service level agreements. In: 2005 IEEE International Conference on Services Computing (SCC'05) Vol-1. vol. 2, pp. 49–56 (2005)
4. Kleehaus, M., Uludag, Ö., Schäfer, P., Matthes, F.: MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments, pp. 148–162 (Jun 2018). https://doi.org/10.1007/978-3-319-92901-9_14
5. Mohamed, A., Zulkernine, M.: On failure propagation in component-based software systems. In: 2008 The Eighth International Conference on Quality Software. pp. 402–411 (2008). https://doi.org/10.1109/QSIC.2008.46
6. Richardson, C.: Microservices Patterns: With examples in Java. Manning Publications (2018)
7. Speth, S.: Semi-automated Cross-Component Issue Management and Impact Analysis. In: Proceedings of 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1090–1094. IEEE, IEEE (Nov 2021)
8. Speth, S., Becker, S., Breitenbücher, U.: Cross-Component Issue Metamodel and Modelling Language. In: Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021). SciTePress (May 2021)
9. Speth, S., Stieß, S., Becker, S.: A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis. In: Companion Proceedings of 19^{th} IEEE International Conference on Software Architecture (ICSA-C 2022). IEEE (2022)