SOA and the Button Problem

Sung-Shik Jongmans[1,2], Arjan Lamers[1,3], and Marko van Eekelen[1,4]

[1] Department of Computer Science, Open University of the Netherlands
[2] CWI, Netherlands Foundation of Scientific Research Institutes
[3] First8, Nijmegen, the Netherlands
[4] Institute for Computing and Information Sciences, Radboud University Nijmegen

Abstract. Service-oriented architecture (SOA) is a popular architectural style centered around services, loose coupling, and interoperability. A recurring problem in SOA development is the *Button Problem*; how to ensure that whenever a "button is pressed" on some service—no matter what—the performance of other key services remains unaffected? The Button Problem is especially complex to solve in systems that have devolved into hardly comprehensible spaghettis of service dependencies.
In a collaborative effort with industry partner First8, we present the first formal framework to help SOA developers solve the Button Problem, enabling automated reasoning about service sensitivities and candidate refactorings. Our formalization provides a rigorous foundation for a tool that was already successfully evaluated in industrial case studies, and it is built against two unique requirements: "whiteboard level of abstraction" and non-quantitative analysis.

1 Introduction

Context. Service-oriented architecture (SOA) is a popular architectural style centered around services, loose coupling, and interoperability [19].

A recurring problem in SOA development is the *Button Problem*: how to ensure that whenever a "button is pressed" (i.e., an operation is invoked; a resource is requested) on some service—no matter what—the performance of other key services remains unaffected? For instance, increased activity on an accounting service of an e-commerce system should never slow down the front-end service; sales are lost otherwise [1]. The Button Problem occurs in all stages of SOA development, from initial analysis (when dependencies among services are still reasonably well-understood) to final maintenance (when dependencies have often devolved into a hardly comprehensible spaghetti).

To solve the Button Problem, SOA developers need to engage in two kinds of activities: (1) they need to analyze dependencies among services to determine whether or not a service is indeed *sensitive* to button-presses on other services; if so, (2) they need to invent a series of *refactorings* that eliminate the sensitivity, but without changing the system's functional behavior. Especially in cases where services and their dependencies are plentiful, these two activities are challenging to carry out by hand: both service sensitivities and candidate refactorings are easily missed, leading to suboptimal architecture and deployment decisions.

Contribution. In a collaborative effort with industry partner First8, we present the first formal framework to help SOA developers solve the Button Problem, enabling automated reasoning about service sensitivities and candidate refactorings in the form of tool support. Our formalization is built against two unique requirements derived from First8's experience with large enterprise systems (Sect. 2): "whiteboard level of abstraction" and non-quantitative analysis. We provide an extensible core library of refactorings and prove their correctness; this facilitates mechanical exploration of a system's design space toward given insensitivity goals. Our formalization provides a rigorous foundation for a decision support tool (Sect. 2) that we developed and recently demonstrated at ICSOC 2018 [16].

In Sect. 2, we explain the background of this research project. In Sect. 3, we present our formalization of architectures and refactorings. In Sect. 4, we present our formalization of deployments and sensitivities. In Sect. 5, we explain the implementation of our formal framework. In Sect. 6, we discuss related work and future work. *Proofs of theorems appear in a separate technical report* [9].

2 Background

First8. First8 (`https://www.first8.nl`), subsidiary of Conclusion (`https://www.conclusion.nl`), is a software company specialized in custom business-critical systems, including SOA, in all stages of the software life cycle. SOA developers at First8 regularly encounter and struggle with the Button Problem. In general, the industry-wide practice of manually reasoning about service sensitivities and candidate refactorings has three major issues. *First*, it is an intellectually demanding activity that often requires SOA developers to make simplifying assumptions. This leads to imprecise refactoring proposals, which may be more costly, more risky, and less effective than necessary. *Second*, as refactoring proposals are based on experience and best-practices, SOA developers can easily overlook less-intuitive refactorings that may well be most-effective for a given system. *Third*, predicting how multiple refactorings will affect each other is hard.

The aim of this research project is to develop a decision support tool (open source), built on top of a rigorous foundation, that helps SOA developers (First8 or otherwise) solve the Button Problem. Based on extensive experience with large enterprise systems, First8 imposed two unique requirements on the tool and its underlying formalization that give our project a novel position among existing computer-aided software engineering tools (see also Sect. 6):

- **"Whiteboard level of abstraction":** Finding a technical solution to the Button Problem is one thing; convincing business executives that this solution is truly worth pursuing and implementing is a whole different challenge. Decisions are often made in meetings where there is neither time nor expertise on the executives' side to go through all the technical intricacies; instead, high-level whiteboard drawings are the main artifacts to explain service sensitivities and candidate refactorings, their consequences, and their trade-offs. To truly contribute to executives' decision-making, it is therefore imperative

that our tool and its foundation are based on the simple whiteboard-style notation that executives intuitively understand and are accustomed to.
- **Non-quantitative analysis:** Ultimately, every Button Problem is about coarse-grained predictability of performance; it is *never* about reducing latency by x milliseconds, or increasing throughput by y transactions per hour. Although it is possible to try to solve the Button Problem using fine-grained quantitative approaches in terms of absolute latencies and throughputs (e.g., [3,5,13,15,22]), it is excessive (i.e., not the right tool for the job) and impractical. One issue is collecting the measurements to instantiate a quantitative model, which can be cumbersome or even impossible (i.e., if the system has not been deployed yet). Another issue is that measurements are implementation-specific and deployment-specific, and therefore brittle: changes in a service implementation or deployment can greatly impact absolute performance and immediately render a previously instantiated quantitative model obsolete. To solve the Button Problem *effectively*, using automated tool support, a non-quantitative approach is needed.

The Elmo tool. Elmo is the decision support tool that is developed in this research project (open source; `https://bitbucket.org/arjanl/elmo-tool`), recently demonstrated at ICSOC 2018 [16,10] and built on top of the formalization presented in this paper. Leveraging a whiteboard-style notation for architectures and deployments, Elmo's main features are (1) automated non-quantitative analysis of service sensitivities and (2) automated inference of series of candidate refactorings that are guaranteed to be behavior-preserving and achieve given insensitivity goals. If multiple different series of candidate refactorings achieve the specified goals, Elmo automatically computes a comparison of other attributes of the final system designs for the user to inspect. Moreover, Elmo also supports an interactive mode that enables users to manually explore a system's design space by selecting and applying candidate refactorings from a list. (Elmo does not actually carry out refactorings, though; the tool is geared toward providing decision support to solve the Button Problem.)

We successfully evaluated Elmo in two case studies involving systems of First8 clients that suffer(ed) from the Button Problem:

1. In an e-commerce system at an undisclosed client, performance issues arose in key services when the load on seemingly unrelated services increased. First8 was consulted to solve this Button Problem, but Elmo did not yet exist at the time. Due to the sheer size and complexity of the system, the SOA developers involved ultimately proposed a broad, coarse-grained refactoring approach that affected the whole system; they were unable to manually find a more targeted series of refactorings to solve the problem more locally. The project revealed the need for a decision support tool to deal with this complexity.
   Recently, we modeled the system in Elmo and automatically found a much more localized series of refactorings that achieves the same goals. Moreover, SOA developers that worked on the project are of the opinion that if Elmo had existed at the time of the project, this would have resulted in performance improvements much earlier in the process and with more confidence.

2. JoinData is a digital highway for farm-generated data, used nation-wide in the Netherlands. It allows for data exchange in the agricultural sector. For example, milking-robots on the farm, animal feed suppliers, or soil laboratories can exchange information with accountancy firms, governmental organisations, or farm management systems. Due to expected growth, the scalability of the messaging component, called EDI-Circle, needed to improve by solving the Button Problem of one of its constituent services.
   We compared (i) the manual analysis and proposed course of action by the lead architect of EDI-Circle with (ii) Elmo's automated analysis. Whereas the architect proposed "to change the whole system, since everything is connected", Elmo proposed a much more localized series of refactorings.

Details of case study 1 are protected by NDA; details of case study 2 are in [16].

In the rest of this paper, we present the rigorous foundation on top of which Elmo is built. We shall formalize systems at the abstraction level of their architectures and deployments. Refactorings are subsequently defined over formal architecture models; sensitivities are derived from formal deployment models.

3 Architectures & Refactorings

3.1 Architecture Models

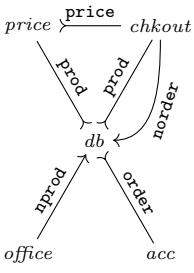
Our formalization of architectures closely follows the proven whiteboard-style notation used by First8's SOA developers to effectively communicate with clients and business executives. The notation comprises graphical diagrams where services are drawn as nodes and calls between services as edges. Services are annotated with the *types* of information they *produce* and *consume*. Calls come in two flavors: *pushes* and *pulls*. A push by service s_1 to service s_2 entails a single communication from s_1 to s_2 (there is no subsequent acknowledgment from s_2 to s_1[1]); a pull by s_1 from s_2 entails a request for information from s_1 to s_2, and a subsequent response from s_2 to s_1. We formalize these diagrams as architecture models. Let $\mathbb{S}$ denote the set of all services, ranged over by s, and let $\mathbb{T}$ denote the set of all types of information, ranged over by t.

Definition 1. *An* ***architecture model*** A *is a tuple* $(S, T, \Pi, \Gamma, \longrightarrow, \longrightarrow\!\!\!\!\prec)$ *where:*

- $S \subseteq \mathbb{S}$ *and* $T \subseteq \mathbb{T}$ *denote sets of services and types;*
- $\Pi, \Gamma : T \to 2^S$ *denote indexed sets of producers and consumers;*
- $\longrightarrow, \longrightarrow\!\!\!\!\prec \; : T \to 2^{S \times S}$ *denote indexed push and pull relations such that* $(s_1, s_2) \in \longrightarrow(t)$ *implies* $s_1 \neq s_2$*, and* $(s_1, s_2) \in \longrightarrow\!\!\!\!\prec(t)$ *implies* $s_1 \neq s_2$*.*

$\mathbb{A}\text{rch}$ *denotes the set of all architecture models.*

[1] In terms of the Osi transport layer, Tcp/Ip packets involved in a push *are* acknowledged (as part of the Tcp/Ip protocol), but this is at a lower level of abstraction.



$\Pi^{\texttt{nprod}} = \{office\}$
$\Pi^{\texttt{norder}} = \{chkout\}$
$\Pi^{\texttt{prod}} = \{db\}$
$\Pi^{\texttt{order}} = \{db\}$
$\Pi^{\texttt{price}} = \{price\}$
$\Gamma^{\texttt{nprod}} = \{db\}$
$\Gamma^{\texttt{norder}} = \{db\}$
$\Gamma^{\texttt{prod}} = \{price, chkout\}$
$\Gamma^{\texttt{order}} = \{acc\}$
$\Gamma^{\texttt{price}} = \{chkout\}$

Fig. 1: Example architecture model

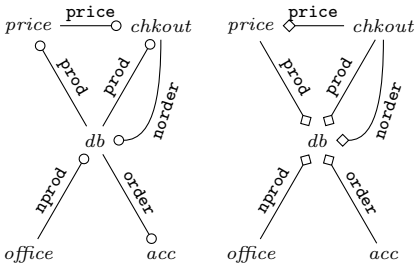


Fig. 2: Example directions/initiatives

In words, $s \in \Pi(t)$ and $s \in \Gamma(t)$ mean that service s respectively produces and consumes information of type t (the utility of these sets becomes clear when we define well-formedness, shortly); we write Π^t and Γ^t instead of $\Pi(t)$ and $\Gamma(t)$. In words, $(s_1, s_2) \in \longrightarrow(t)$ and $(s_1, s_2) \in \longrightarrow\!\!\!\!\prec(t)$ mean that service s_1 respectively pushes and pulls information of type t to and from service s_2; we write $s_1 \xrightarrow{t} s_2$ and $s_1 \xrightarrow{t}\!\!\!\!\prec\ s_2$ instead of $(s_1, s_2) \in \longrightarrow(t)$ and $(s_1, s_2) \in \longrightarrow\!\!\!\!\prec(t)$. The *domain* of an architecture model A is its set of services, denoted by $\mathrm{Dom}(A)$.

Example 1. Figure 1 shows an architecture model for a webshop system; it is a simplified version of the e-commerce system discussed in Sect. 2.

The database service, called *db*, manages information about products and orders. The front-end service, called *chkout*, is used by customers to order products; it calls the database service to pull product information and push new orders, while it pulls from a pricing service, called *price*, for calculating final prices (including additional fees and transport costs). The accounting service, called *acc*, checks if orders have been paid for; it calls the database service to pull order information. Finally, the back-office service, called *office*, maintains the product catalog; it calls the database service to push updated product information.

We note that we distinguish between new order/product information (`nprod` and `norder`), produced by *chkout*/*office*, and existing order/product information (`prod` and `order`), produced (i.e., "owned") by *db*. □

Architecture models (Defn. 1) specify precisely the *direction* (i.e., from pusher to "pushee", but from "pullee" to puller) and the *initiative* (i.e., pushers and pullers; services that start information flows) of information flows; they abstract from call specifics (e.g., operations that are invoked; resources that are accessed), quantitative aspects of communication (e.g., call frequencies; latency; throughput), and transport characteristics (e.g., synchronous vs. asynchronous; reliable vs. lossy; unordered vs. order-preserving). Direction and initiative serve key purposes in our work: in this section, we use direction to reason about candidate refactorings; in the next section, we use initiative to reason about service sensitivities. We elicit these notions formally as follows. Let $f_1 \uplus f_2 = \{x \mapsto f_1(x) \mid x \in X_1 \setminus X_2\} \cup \{x \mapsto f_2(x) \mid x \in X_2 \setminus X_1\} \cup \{x \mapsto f_1(x) \cup f_2(x) \mid x \in X_1 \cap X_2\}$ denote the pointwise union of functions $f_1 : X_1 \to 2^{Y_1}$ and $f_2 : X_2 \to 2^{Y_2}$.

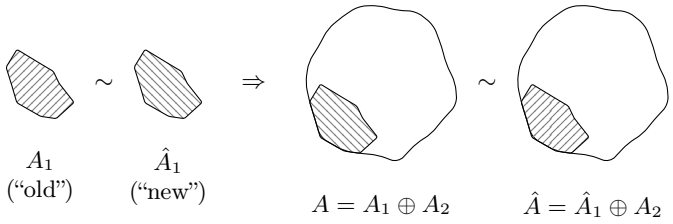


Fig. 3: Refactoring framework

Definition 2. $\multimap\!\circ, \multimap\!\diamond : \mathbb{A}\mathrm{rch} \to (\mathbb{T} \to 2^{\mathbb{S}\times\mathbb{S}})$ *denote the (doubly) indexed* ***direction and initiative relations*** *defined by the following equations:*

$$\multimap\!\circ((S,T,\Pi,\Gamma,\longrightarrow,\multimap\!\prec)) = \left\{ t \mapsto \longrightarrow(t) \cup \multimap\!\prec(t)^{-1} \;\middle|\; t \in T \right\}$$
$$\multimap\!\diamond((S,T,\Pi,\Gamma,\longrightarrow,\multimap\!\prec)) = \longrightarrow \uplus \multimap\!\prec$$

In words, $(s_1, s_2) \in \multimap\!\circ(A)(t)$ and $(s_1, s_2) \in \multimap\!\diamond(A)(t)$ means that flow of information of type t is directed and initiated from service s_1 to service s_2 in architecture model A; we write $s_1 \overset{t}{\multimap\!\circ}_A s_2$ and $s_1 \overset{t}{\multimap\!\diamond}_A s_2$ instead of $(s_1, s_2) \in \multimap\!\circ(A)(t)$ and $(s_1, s_2) \in \multimap\!\diamond(A)(t)$. Fig. 2 exemplifies these relations for Fig. 1.

An architecture model is *well-formed* if every flow of information of type t starts at a producer of t (i.e., information should not emerge out of nowhere) and ends at a consumer of t (i.e., information should not be discarded unused). Formally, if $A = (S,T,\Pi,\Gamma,\longrightarrow,\multimap\!\prec)$ and $s_1 \overset{t}{\multimap\!\circ}_A s_2$, then there exist services s_p and s_c such that: $\Pi^t \ni s_\mathrm{p} \overset{t}{\multimap\!\circ}^*_A s_1$ and $s_2 \overset{t}{\multimap\!\circ}^*_A s_\mathrm{c} \in \Gamma^t$. Well-formedness is an important sanity condition that models need to satisfy; it catches modeling inconsistencies and redundancies regarding information availability (which is also why producers/consumers are explicit elements of the model and not derived).

3.2 Refactoring Framework

We define a rigorous refactoring framework in terms of *composition* ($\oplus$) and *equivalence* ($\sim$) of architecture models (Fig. 3). The idea is to represent an architecture A as the composition of an "old part" A_1 and a "remaining part" A_2 (formally: $A = A_1 \oplus A_2$). Refactoring, then, amounts to substituting the old part with an equivalent "new part" $\hat{A}_1$ (formally: $A_1 \sim \hat{A}_1$). If the equivalence is in fact a *congruence* for composition, substitution of equivalent parts is guaranteed to yield equivalent wholes, which means that all existing information flows are preserved by substitution and no spurious new ones are introduced. This congruence property is pivotal: because of it, to show that a refactoring is correct, we need to prove only the equivalence of the old part and the new part, while we can safely ignore the remaining part. We now explain the details.

To compose architecture models A_1 and A_2, we "glue" them together on their shared services; through these services, information can subsequently flow from A_1 to A_2 and back, in accordance with the original push and pull relations. Such

composition of architecture models corresponds roughly to union of graphs with overlapping vertex sets but disjoint edge sets.

Definition 3. $\oplus : \mathbb{A}\mathrm{rch} \times \mathbb{A}\mathrm{rch} \to \mathbb{A}\mathrm{rch}$ *denotes the* ***composition function*** *defined by the following equation:*

$$A_1 \oplus A_2 = (S_1 \cup S_2, T_1 \cup T_2, \Pi_1 \uplus \Pi_2, \Gamma_1 \uplus \Gamma_2, \longrightarrow_1 \uplus \longrightarrow_2, \longrightarrow\!\!\!\!\prec_1 \uplus \longrightarrow\!\!\!\!\prec_2)$$
$$\text{where } A_i = (S_i, T_i, \Pi_i, \Gamma_i, \longrightarrow_i, \longrightarrow\!\!\!\!\prec_i)$$

The following theorem states that composition of architecture models preserves the direction of information flows.

Theorem 1. $\longrightarrow\!\!\!\!\circ_{A_1 \oplus A_2} = \longrightarrow\!\!\!\!\circ_{A_1} \uplus \longrightarrow\!\!\!\!\circ_{A_2}$

Two architecture models are equivalent iff the direction of every flow of information in the one can be mimicked in the other, including production and consumption of information, and vice versa. We note that we do not require mimicry of initiative; the idea is that it does not matter which service initiates sharing of information, so long as all information reaches the right services.

Definition 4. $\preceq, \sim \; : \; 2^{\mathbb{S}\times\mathbb{S}} \to 2^{\mathbb{A}\mathrm{rch}\times\mathbb{A}\mathrm{rch}}$ *denote the indexed* ***preorder and equivalence relations*** *defined by the following equations:*

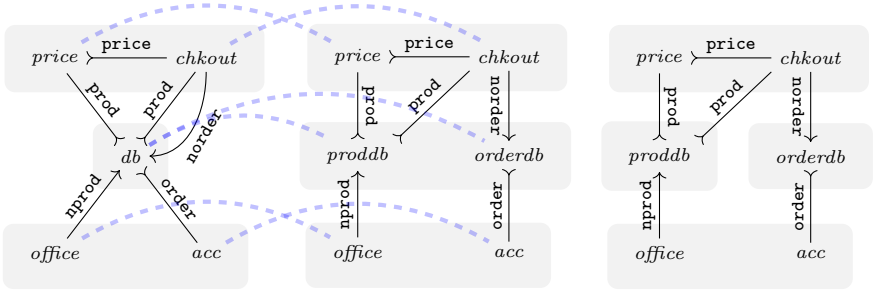
$$\preceq(R) = \left\{ (A, \hat{A}) \;\middle|\; \begin{array}{l} \forall t, s, s'. \left[s \overset{t}{\longrightarrow\!\!\!\!\circ}_A s' \;\Rightarrow\; \exists \hat{s}, \hat{s}'. \left[\hat{s} \overset{t}{\longrightarrow\!\!\!\!\circ}_{\hat{A}} \hat{s}' \;\wedge\; s \; R \; \hat{s} \;\wedge\; s' \; R \; \hat{s}' \right] \right] \\ \wedge\; \forall t, s. \left[s \in \Pi^t \;\Rightarrow\; \exists \hat{s}. \left[\hat{s} \in \hat{\Pi}^t \;\wedge\; s \; R \; \hat{s} \right] \right] \\ \wedge\; \forall t, s. \left[s \in \Gamma^t \;\Rightarrow\; \exists \hat{s}. \left[\hat{s} \in \hat{\Gamma}^t \;\wedge\; s \; R \; \hat{s} \right] \right] \\ \wedge\; A = (S, T, \Pi, \Gamma, \longrightarrow, \longrightarrow\!\!\!\!\prec) \;\wedge\; \hat{A} = (\hat{S}, \hat{T}, \hat{\Pi}, \hat{\Gamma}, \overset{\sim}{\longrightarrow}, \overset{\sim}{\longrightarrow\!\!\!\!\prec}) \end{array} \right\}$$
$$\sim(R) = \preceq(R) \cap \preceq(R^{-1})^{-1}$$

In words, $(A, \hat{A}) \in \preceq(R)$ means that relation R associates every service s in A with a set of services $\hat{S} = \{\hat{s} \mid s \; R \; \hat{s}\}$ in $\hat{A}$ *that collectively*[2] *simulate* s (i.e., every information flow from s to some service s' in A can be mimicked as an information flow from some service $\hat{s} \in \hat{S}$ to some service $\hat{s}'$ in $\hat{A}$; every information production or consumption by s can be mimicked as an information production or consumption by some service $\hat{s} \in \hat{S}$). In words, $(A, \hat{A}) \in \sim(R)$ means that services in A and in $\hat{A}$ simulate each other under the same relation R. We write $A \preceq_R \hat{A}$ and $A \sim_R \hat{A}$ instead of $(A, \hat{A}) \in \preceq(R)$ and $(A, \hat{A}) \in \sim(R)$.

Example 2. Figures 4a and 4b show two equivalent architecture models of the example webshop system (Exmp. 1), before and after refactoring; we discuss Fig. 4c, the gray boxes around services, and the parenthetical mentioning of "deployment models" in the caption in Sect. 4.

Architecture model A in Fig. 4a is the original (cf. Fig. 1). Architecture model $\hat{A}$ in Fig. 4b results from "splitting" service *db* in A into two new services: one that

[2] Individual services in $\hat{S}$ may contribute only partially to the simulation (see also Exmp. 2). This is where our definition of simulation differs significantly from the classical one in concurrency theory (e.g., [17]). It is also why $s \; R \; \hat{s}$ appears as a conjunct on the right-hand side of the implication instead of on the left-hand side.



(a) Original A (b) After split $\hat{A}$ (c) After redeployment

Fig. 4: Equivalent architecture models (and deployment models) of the example webshop systems before and after refactoring. Blue dashed lines indicate the simulation relation; sets of producers and consumers are omitted to save space.

stores only product information, and one that stores only order information. To see that A and $\hat{A}$ are equivalent, observe that `prod` information flows from *db* to services *chkout* and *price* in A, while `prod` information flows from service *proddb* to *chkout* and *price* in $\hat{A}$. Thus, *db* in A is partially simulated by *proddb* in $\hat{A}$; likewise, with respect to `order` information flows, *db* in A is partially simulated by service *orderdb* in $\hat{A}$. Thus, *db* in A is collectively simulated by *proddb* and *orderdb* in $\hat{A}$. Similarly, we can argue that $\hat{A}$ is simulated by A. □

Example 3. To further illustrate (the intricacies of) Defn. 4, suppose well-formed architecture A precisely consists of $\Pi^t \ni s \xrightarrow{t} s' \xrightarrow{t} s'' \in \Gamma^t$, while well-formed architecture $\hat{A}$ precisely consists of $\Pi^t \ni s \xrightarrow{t} s'_{\mathrm{a}} \in \Gamma^t$ and $\Pi^t \ni s'_{\mathrm{b}} \xrightarrow{t} s'' \in \Gamma^t$. These architectures are *not* equivalent: no relation R exists such that $A \sim_R \hat{A}$. Notably, $A \not\sim_{R^\dagger} \hat{A}$ for $R^\dagger = \{(s,s),(s',s'_{\mathrm{a}}),(s',s'_{\mathrm{b}}),(s'',s'')\}$, because s'_{a} (resp. s'_{b}) in $\hat{A}$ is consumer (resp. producer), but s' in A is not. Also, $A \not\sim_{R^\ddagger} \hat{A}$ for $R^\ddagger = \{(s,s),(s,s'_{\mathrm{b}}),(s'',s'_{\mathrm{a}}),(s'',s'')\}$, because s' is missing from $R^\ddagger$. This also shows that well-formedness does not imply production/consumption mimicry.

But, $A \sim_{R^\dagger} \hat{A}$ *does hold* after updating A such that $s' \in \Pi^t \cap \Gamma^t$. In that case, splitting s' into s'_{a} and s'_{b} means the consumption and production responsibilities of s' are divided over two new services; this can be perfectly fine in practice. □

The following theorem states that the equivalence relation $\sim$ (Defn. 4) is a congruence relation for the composition operation $\oplus$ (Defn. 3). To prove the theorem, we need additional assumptions beside equivalence of the parts. These additional assumptions state that *after substitution*, the services on the "boundary" between the old/new parts and the remaining part (set S_B in the theorem) must be indistinguishable from those *before substitution* (in terms of their names and information flows). In other words, the interface must remain the same: services on the boundary may not be renamed, added, or removed by a refactoring.

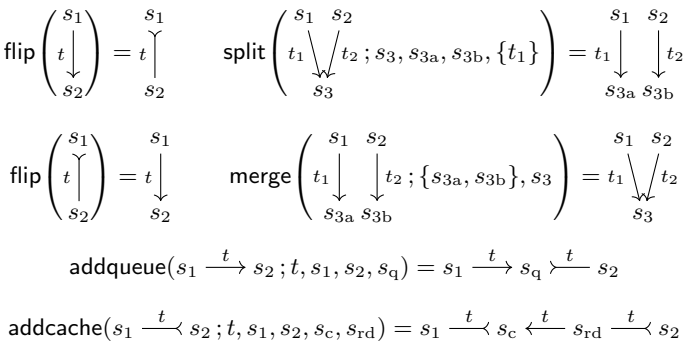


Fig. 5: Basic instances of example refactorings

Theorem 2.

$$\left[\begin{array}{l} A_1 \sim_{R_1} \hat{A}_1 \\ \wedge\ S_{\mathrm{B}} = \mathrm{Dom}(A_1) \cap \mathrm{Dom}(A_2) \\ \wedge\ S_{\mathrm{B}} = \mathrm{Dom}(\hat{A}_1) \cap \mathrm{Dom}(A_2) \\ \wedge\ \forall s, \hat{s}.\big[\big[s\ R_1\ \hat{s}\ \wedge\ s \in S_B\big]\ \Rightarrow\ s = \hat{s}\big] \\ \wedge\ \forall s, \hat{s}.\big[\big[s\ R_1\ \hat{s}\ \wedge\ \hat{s} \in S_B\big]\ \Rightarrow\ s = \hat{s}\big] \end{array}\right] \Rightarrow\ \exists R.\big[A_1 \oplus A_2 \sim_R \hat{A}_1 \oplus A_2\big]$$

3.3 Core Library of Refactorings

Now, every refactoring in our framework (Fig. 3) is defined by a predicate–function pair (P, f): predicate P identifies (sub)architectures that can take on the role of A_1 (the old part), while function f describes the transformation of A_1 into $\hat{A}_1$ (the new part). An *instance* of a refactoring, then, is the transformation of a concrete A_1 that satisfies P into $\hat{A}_1$ according to f. We call refactoring (P, f) *correct* if, for all A_1, satisfaction of P by A_1 implies that A_1 and $f(A_1)$ are equivalent. Subsequently, Thm. 2 ensures that a correct refactoring for A_1 can safely be applied in any architecture that contains A_1.

We defined a core library of provably correct refactorings: *Flip*, *Split*, *Merge*, *AddQueue*, and *AddCache*. These refactorings were selected to form a minimal set of primitive building blocks to support our two case studies (Sect. 2); due to the generality of our framework, the core library can straightforwardly be extended in future work, by need. Figure 5 shows basic instances of these refactorings; notationally, we use a semicolon to distinguish the old architecture to which a refactoring is applied from additional information that is used to compute the refactoring. Refactoring *Flip* converts pushes between corresponding "reverse-pulls" and vice versa. Refactoring *Split* divides the responsibilities of a single old service s_3 over multiple new services $s_{3\mathrm{a}}$ and $s_{3\mathrm{b}}$ (practically, such splitting is usually subject to additional constraints, such as information dependencies, which can be manually added as model annotations in the implementation; Sect. 5, footnote 4). Dually, refactoring *Merge* combines the responsibilities

Let $A = (S, \Pi, \Gamma, \longrightarrow, \longrightarrow\!\!\!\!\prec)$. Predicates $\mathsf{Flip?}, \mathsf{Split?}, \mathsf{Merge?}, \mathsf{AddQueue?}, \mathsf{AddCache?}$ are the smallest relations induced by the following rules:

$$\frac{}{\mathsf{Flip?}(A)} \qquad \frac{s \in S \ \wedge \ s_1, s_2 \notin S}{\mathsf{Split?}(A; s, s_1, s_2, T_1)} \qquad \frac{S' \subseteq S \ \wedge \ s \notin S}{\mathsf{Merge?}(A; S', s)}$$

$$\frac{s_1 \xrightarrow{t} s_2 \ \wedge \ s \notin S}{\mathsf{AddQueue?}(A; t, s_1, s_2, s)} \qquad \frac{s_1 \xrightarrow{t}\!\!\!\!\prec\ s_2 \ \wedge \ s, s' \notin S}{\mathsf{AddCache?}(A; t, s_1, s_2, s, s')}$$

Fig. 6: Predicates of refactorings in the core library

of multiple old services $s_{3\mathrm{a}}$ and $s_{3\mathrm{b}}$ into a single new service s_3. Refactoring *AddQueue* introduces a special service s_{q} to replace a push from service s_1 to service s_2; the idea is that "producer" s_1 now pushes information to "queue" s_{q} (instead of directly to s_2), while "consumer" s_2 pulls that information from s_{q} (at its own pace, independent of s_1). Refactoring *AddCache* introduces special services s_{c} and s_{rd} to replace a pull from service s_1 to service s_2; the idea is that "consumer" s_1 pulls information from "cache" s_{c}, which is eagerly filled through pushes from "reader" s_{rd}, which gets the information by pulling from "producer" s_2.

Let $X[y/Y]$ denote the substitution in X of element y for every element from set Y (i.e., $X[y/Y] = X$ if $X \cap Y = \emptyset$, and $X[y/Y] = (X \setminus Y) \cup \{y\}$ otherwise), and let $\circ : 2^{X \times Y} \times 2^{Y \times Z} \to 2^{X \times Z}$ denote relational composition. Figures 6 and 7 show the predicates and functions that formally define the refactorings in the core library. The following theorem states their correctness.

Theorem 3.

- $\mathsf{Flip?}(A) \ \Rightarrow \ \exists R.\big[A \sim_R \mathsf{flip}(A)\big]$
- $\mathsf{Split?}(A; s, s_1, s_2, T_1) \ \Rightarrow \ \exists R.\big[A \sim_R \mathsf{split}(A; s, s_1, s_2, T_1)\big]$
- $\mathsf{Merge?}(A; S, s) \ \Rightarrow \ \exists R.\big[A \sim_R \mathsf{merge}(A; S, s)\big]$
- $\mathsf{AddQueue?}(A; s_{\mathrm{q}}) \ \Rightarrow \ \exists R.\big[A \sim_R \mathsf{addqueue}(A; s_{\mathrm{q}})\big]$
- $\mathsf{AddCache?}(A; s_{\mathrm{c}}, s_{\mathrm{rd}}) \ \Rightarrow \ \exists R.\big[A \sim_R \mathsf{addcache}(A; s_{\mathrm{c}}, s_{\mathrm{rd}})\big]$

Together, Thms. 2 and 3 support the refactoring framework shown in Fig. 3. We note, though, that to apply Thm. 2 with *Split* and *Merge* (i.e., to satisfy the boundary condition), not only the split/merged service s must be in the old/new parts, but also the services that s calls and those that call s.

4 Deployments & Sensitivities

The whiteboard-style architecture models and rigorous refactoring framework presented in Sect. 3 offer a formal means of defining and reasoning about (the correctness of) refactorings. However, the formalism so far does not tell us which refactorings are "good" and which ones are "bad"; what is missing is a mechanism to evaluate the effectiveness of a refactoring. In this section, we define non-quantitative sensitivity indicators based on which SOA developers can make informed choices between candidate refactorings to solve the Button Problem.

Let $A = (S, \Pi, \Gamma, \longrightarrow, \longrightarrow\!\!\!\!\prec)$. Functions $\mathsf{flip}, \mathsf{split}, \mathsf{merge}, \mathsf{addqueue}, \mathsf{addcache}$ are defined by the following equations:

$$\mathsf{flip}(A) = (S, \Pi, \Gamma, \{t \mapsto (\overset{t}{\longrightarrow\!\!\!\!\prec})^{-1} \mid t \in \mathbb{T}\}, \{t \mapsto (\overset{t}{\longrightarrow})^{-1} \mid t \in \mathbb{T}\})$$

$$\begin{aligned}
\mathsf{split}(A; s, s_1, s_2, T_1) &= ((S \setminus \{s\}) \cup \{s_1, s_2\}, \hat{\Pi}, \hat{\Gamma}, \overset{\wedge}{\longrightarrow}, \overset{\wedge}{\longrightarrow\!\!\!\!\prec}) \\
\hat{\Pi} &= \{t \mapsto \Pi^t[s_1/\{s\}] \mid t \in T_1\} \cup \{t \mapsto \Pi^t[s_2/\{s\}] \mid t \notin T_1\} \\
\hat{\Gamma} &= \{t \mapsto \Gamma^t[s_1/\{s\}] \mid t \in T_1\} \cup \{t \mapsto \Gamma^t[s_2/\{s\}] \mid t \notin T_1\} \\
\overset{\wedge}{\longrightarrow} &= \{t \mapsto \{(s_1, s)\} \circ \overset{t}{\longrightarrow} \circ \{(s, s_1)\} \mid t \in T_1\} \cup \\
&\quad \{t \mapsto \{(s_2, s)\} \circ \overset{t}{\longrightarrow} \circ \{(s, s_2)\} \mid t \notin T_1\} \\
\overset{\wedge}{\longrightarrow\!\!\!\!\prec} &= \{t \mapsto \{(s_1, s)\} \circ \overset{t}{\longrightarrow\!\!\!\!\prec} \circ \{(s, s_1)\} \mid t \in T_1\} \cup \\
&\quad \{t \mapsto \{(s_2, s)\} \circ \overset{t}{\longrightarrow\!\!\!\!\prec} \circ \{(s, s_2)\} \mid t \notin T_1\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{merge}(A; S', s) &= ((S \setminus S') \cup \{s\}, \hat{\Pi}, \hat{\Gamma}, \overset{\wedge}{\longrightarrow}, \overset{\wedge}{\longrightarrow\!\!\!\!\prec}) \\
\hat{\Pi} &= \{t \mapsto \Pi^t[s/S'] \mid t \in \mathbb{T}\} \\
\hat{\Gamma} &= \{t \mapsto \Gamma^t[s/S'] \mid t \in \mathbb{T}\} \\
\overset{\wedge}{\longrightarrow} &= \{t \mapsto (s \times S') \circ \overset{t}{\longrightarrow} \circ (S' \times s) \mid t \in \mathbb{T}\} \\
\overset{\wedge}{\longrightarrow\!\!\!\!\prec} &= \{t \mapsto (s \times S') \circ \overset{t}{\longrightarrow\!\!\!\!\prec} \circ (S' \times s) \mid t \in \mathbb{T}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{addqueue}(A; t, s_1, s_2, s) &= (S \cup \{s\}, \Pi, \Gamma, \overset{\wedge}{\longrightarrow}, \overset{\wedge}{\longrightarrow\!\!\!\!\prec}) \\
\overset{\wedge}{\longrightarrow} &= (\longrightarrow \setminus \{t \mapsto \overset{t}{\longrightarrow}\}) \cup \{t \mapsto (\overset{t}{\longrightarrow} \setminus \{(s_1, s_2)\}) \cup \{(s_1, s)\}\} \\
\overset{\wedge}{\longrightarrow\!\!\!\!\prec} &= (\longrightarrow\!\!\!\!\prec \setminus \{t \mapsto \overset{t}{\longrightarrow\!\!\!\!\prec}\}) \cup \{t \mapsto \overset{t}{\longrightarrow\!\!\!\!\prec} \cup \{(s_2, s)\}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{addcache}(A; t, s_1, s_2, s, s') &= (S \cup \{s, s'\}, \Pi, \Gamma, \overset{\wedge}{\longrightarrow}, \overset{\wedge}{\longrightarrow\!\!\!\!\prec}) \\
\overset{\wedge}{\longrightarrow} &= (\longrightarrow \setminus \{t \mapsto \overset{t}{\longrightarrow}\}) \cup \{t \mapsto \overset{t}{\longrightarrow} \cup \{(s', s)\}\} \\
\overset{\wedge}{\longrightarrow\!\!\!\!\prec} &= (\longrightarrow\!\!\!\!\prec \setminus \{t \mapsto \overset{t}{\longrightarrow\!\!\!\!\prec}\}) \cup \\
&\quad \{t \mapsto (\overset{t}{\longrightarrow\!\!\!\!\prec} \setminus \{s_1, s_2\}) \cup \{(s_1, s), (s', s_2)\}\}
\end{aligned}$$

Fig. 7: Functions of refactorings in the core library

Example 4. To illustrate core concepts, we shall continue to develop the example webshop system (Exmp. 1) It actually suffers from exactly the same Button Problem as the e-commerce system on which our simplified version is based.

Specifically, services *chkout* and *price* (in the system as modeled in Fig. 4a) are sensitive to button-presses on service *acc*: once *acc* starts checking whether orders have been paid for, the performance of *chkout* and *price* decreases, as service *db* is unable to process the additional calls from *acc* without affecting the calls from *chkout* and *price*. Checking payment statuses is, however, only a low-priority task—it does not matter whether it happens immediately or in a few hours—and it should definitely not hinder the high-priority front-end of the system (which directly affects business). Refactoring the system to make *chkout* and *price* insensitive to *acc* is therefore an important improvement. □

We start by observing that the sensitivity of a service to button-presses on other services does not depend solely on its incoming push and pull calls, but also on the machine on which it is deployed: if two architecturally independent services are deployed on the same machine, an increased load on the one will affect the performance of the other. To reason about service sensitivities, we

therefore need to take into account deployments as well. Let $\mathbb{M}$ denote the set of all machines, ranged over by M.

Definition 5. *A deployment model D is a tuple $(A, M, \mathcal{M})$ where:*

- $A \in \mathbb{A}\text{rch}$ *denotes an architecture model;*
- $M \subseteq \mathbb{M}$ *denotes a set of machines;*
- $\mathcal{M} : \text{Dom}(A) \to M$ *denotes a service–machine allocation.*

$\mathbb{D}$ *denotes the set of all deployment models.*

Example 5. Reconsider Fig. 4; it actually shows deployment models, where gray boxes around services represent machines. Thus, in Figs. 4a and 4b, there are three machines (from top to bottom: a front-end machine, a database machine, and an administration machine), whereas in Fig. 4c, there are four machines. □

Based on a deployment model of a system, we can compute two non-quantitative indicators that we shortly use to formalize sensitivity: *stress* and *delay*. The stress of a service is a non-quantitative abstraction of the number of incoming calls that it needs to process. The higher the number of calls, the higher the stress of the service and the lower its performance. The delay of a service is a non-quantitative abstraction of the number of outgoing pulls whose processing (by other services) it needs to await. The higher the number of pulls, the higher the delay of the service and the lower its performance. The *stress set* of a service s contains the services that affect the stress of s (including itself): if the stress of a service in its stress set increases, then so does the stress of s. The *delay set* of a service s contains the services that affect the delay of s.

Definition 6. $\mathsf{Stress}, \mathsf{Delay} : \mathbb{D} \times \mathbb{S} \to 2^{\mathbb{S}}$ *denote the indexed* ***stress and delay sets*** *defined by the following equations:*

$$\begin{aligned} \mathsf{Stress}(D, s) &= \{s\} \cup \bigcup\{\mathsf{Stress}(D, s') \mid s' \xrightarrow{t}\!\!\diamond_A s \ \vee \ \mathcal{M}(s') = \mathcal{M}(s)\} \\ \mathsf{Delay}(D, s) &= \bigcup\{\mathsf{Stress}(D, s') \cup \mathsf{Delay}(D, s') \mid s \xrightarrow{t}\!\!\prec s'\} \end{aligned}$$

where $D = (A, M, \mathcal{M})$ and $A = (S, T, \Pi, \Gamma, \longrightarrow, \longrightarrow\!\!\!\!\prec)$.

Note that the delay set of a service s contains the stress set of every service s' from which s pulls information. This is because the services in the stress set of s' may negatively affect the rate at which s' can process pulls by s: if the services in the stress set of s' heavily stress s', then this rate goes down.

We can now formalize (in)sensitivity to button-presses as follows:

- If service s_1 is affected by service s_2 regardless of s_1's calls to s_2, then s_1 is *forcibly sensitive* to s_2 (i.e., s_1 is forcibly sensitive to s_2 if s_2 stresses s_1).
- If service s_1 is affected by service s_2 because s_1 requires information from s_2 by means of a pull, then s_1 is *voluntarily sensitive* to s_2 (i.e., s_1 is voluntarily sensitive to s_2 if s_2 delays s_1).
- If service s_1 is unaffected by service s_2, it is *insensitive* to s_2.

Definition 7. $\mathcal{I}, \mathcal{V}, \mathcal{F} : \mathbb{D} \to \mathbb{S} \times \mathbb{S}$ *denote the indexed* ***sensitivity relations*** *defined by the following equations:*

$$\begin{aligned}\mathcal{F}(D) &= \{(s, s') \mid s' \in \mathsf{Stress}(D, s)\} \\ \mathcal{V}(D) &= \{(s, s') \mid s' \notin \mathsf{Stress}(D, s) \ \wedge \ s' \in \mathsf{Delay}(D, s)\} \\ \mathcal{I}(D) &= \{(s, s') \mid s' \notin \mathsf{Stress}(D, s) \ \wedge \ s' \notin \mathsf{Delay}(D, s)\}\end{aligned}$$

Remark: $\big\{\mathcal{F}(D), \mathcal{V}(D), \mathcal{I}(D)\big\}$ *partitions* $\mathrm{Dom}(A) \times \mathrm{Dom}(A)$ *for* $D = (A, M, \mathcal{M})$.

Example 6. Recall from Exmp. 4 that front-end services *chkout* and *price* suffer from the Button Problem in the example webshop system as modeled in Fig. 4a. We shall apply two changes to alleviate this problem, but first, we show that the deployment model in Fig. 4a indeed confirms these undesirable sensitivities.

Let D denote the deployment model in Fig. 4a. Because services *chkout* and *price* are deployed on the same machine (and because they receive no external calls), their stress set under D is $\{chkout, price\}$. However, because *chkout* and *price* both pull from service *db*, their delay set contains all services that stress *db*, including service *acc*. Thus, $acc \in \mathsf{Delay}(D, chkout)$ and $acc \in \mathsf{Delay}(D, price)$: according to the model high-priority *chkout* and *price* are both voluntarily sensitive to low-priority *acc*. Intuitively, if service *acc* pulls intensely from service *db* (increasing the stress of *db*), the rate at which *db* can process pulls by *chkout* and *price* is negatively affected (increasing the delay of *chkout*).

The first change is the application of refactoring *Split* to divide the responsibilities of existing service *db* over new services *proddb* and *orderdb*; let $\hat{D}$ denote the resulting deployment model in Fig. 4b (Exmp. 2). Intuitively, this refactoring should make services *chkout* and *price* insensitive to button-presses on service *acc* (because the only pulls they perform are directed to *proddb*, which is architecturally independent of *acc*), but because *proddb* and *orderdb* are still deployed on the same machine, the voluntary sensitivities actually remain: *acc* can still stress *orderdb*, which subsequently affects the processing speed of *proddb*.

The second change is a *redeployment* that puts each of services *proddb* and *orderdb* on its own machine; let $\hat{D}$ denote the resulting deployment model in Fig. 4c. A redeployment is not a refactoring, it does not change information flows among services, and thus it is trivially behavior-preserving; it only changes the service–machine allocation. By redeploying services according to $\hat{D}$, stress is no longer shared between *proddb* and *orderdb*; as a result, services *chkout* and *price* become insensitive to *acc*, solving the Button Problem. Reasoning with sensitivities in this way thus provides a formal justification to refactor. □

5 Implementation

Engine. We now explain how the formalization presented in the previous section provides a rigorous foundation for the Elmo tool. To safeguard a tight correspondence between the tool and its formalization, the lead developer of the tool is closely involved in the formalization as well. Essentially, Elmo's implementation consists of two key components: data structures to store architecture models and deployment models and a reasoning engine. The engine has two capabilities:

1. Computation of stress sets, delay sets, (in)sensitivities, and secondary performance indicators (e.g., network depth)
2. Exploration of a system's design space toward one or more insensitivity goals

These capabilities are invoked in Elmo's two usage modes.

Interactive mode. In interactive mode, after drawing an initial deployment model of the system in Elmo, Capability 1 is invoked to get an overview of services that potentially suffer from the Button Problem. The user can subsequently refactor the model to evaluate and compare manually devised candidate solutions. Interactive mode is particularly suitable to get quick feedback on candidate solutions (e.g., during live meetings with project members to explore the options), without having to work out all details manually, which is laborious and error-prone. It is therefore important that computation of performance indicators is fast. To give an indication, the computation of stress sets, delay sets, and (in)sensitivities in the model of the full e-commerce system (case study 1; Sect. 2), which consists of 60 services with 125 calls, takes less than a second (on regular hardware).

Automatic mode. In automatic mode, if service s suffers from the Button Problem (e.g., found using Capability 1), the user can declaratively formulate a solution to the problem as a set of target insensitivities from s to other services; then, Capability 2 is invoked to let Elmo automatically look for series of refactorings that achieve the specified insensitivity goals by exploring the *design space*.

The design space of a system is essentially a directed graph, where vertices are deployment models, and edges are refactorings (from the core library; Sect. 3.3) and redeployments that transform (the architecture model of) a "source" deployment model into (the architecture model of) a "target" deployment model. To generate a system's design space, starting from an initial deployment model, refactorings are applied and sensitivities are computed for the resulting models to check if the specified insensitivity goals have been achieved (using Capability 1). In this way, the entire design space is generated and exhaustively explored; solutions are reported as soon as they are found, so if a satisfactory one is discovered early, the rest of the search may be user-aborted long before exhaustive exploration is done (it can also be bounded to a fixed depth from the start). We employ a breadth-first exploration policy, as it finds solutions of few refactorings (generally more attractive for businesses) sooner than those of many refactorings (generally more expensive). A similar level of automation to explore a system's design space is very difficult to achieve when quantitative models are used, as it is unclear how to get new quantitative data to instantiate refactored models.

The design space generated from an initial deployment model D_0 is finite: there are finitely many services and calls in D_0, there are only finitely many ways in which refactorings create additional services and calls,[3] and the number

[3] More precisely, the only services that create additional services or calls are *Split*, *AddQueue*, and *AddCache*. The number of times a service can be split is bound by the number of types, while the services and calls added through *AddQueue* and *AddCache* carry annotations that inform Elmo to not refactor them any further.

of machines is bounded by the number of services. As a result, under our formalization, the Button Problem is decidable in the sense that Elmo can exhaustively explore the entire design space for solutions. Design spaces do tend to get very large, though, so even if they can be explored in finite time in theory, it may not always be feasible in practice. As a result, pending future optimizations (see below), Elmo's automatic mode is useful in two scenarios:

- **Live meetings:** A question that typically arises during discussions among project members is whether no "easy solutions" (i.e., those that require few refactorings) are overlooked. In this case, Elmo's automatic mode can be effectively used with an explicit depth bound (i.e., maximum number of refactorings that candidate solutions may consist of), significantly reducing the design space to explore. If an easy solution is subsequently found that was previously overlooked, this is of course valuable information; moreover, if no new easy solution is found, this is valuable information, too, as it gives the project team confidence (*and objective data*) to convince executives that a "hard solution" is fundamentally needed. To give an indication, it takes only ten minutes to automatically explore the design space of the full e-commerce system (case study 1; Sect. 2) up to depth 2.
- **Off-line:** In the absence of short deadlines, Elmo's automatic mode can perfectly be run unrestricted, to fully explore the potentially huge design space. A crucial observation is that the size of the design space is not a modeling artifact, but an inherent characteristic of the problem. *Without tool support, SOA developers just have to plow through it by hand*, which seems infeasible; instead, only the more obvious directions are followed, based on experience and best-practices, leaving a large part of the space unexplored and (potentially better) solutions hidden. Our case studies confirm this (Sect. 2): for both systems, Elmo found better solutions that SOA developers did not find.

We are working on a number of optimizations to reduce the design space wherever possible and speed up the exploration: (1) model annotations to further constrain which candidate solutions are truly acceptable;[4] (2) partial order reduction to prune away commuting refactorings [20]; (3) parallelization.

[4] Elmo may find designs that solve the specified insensitivity goals, but that are still unacceptable to SOA developers due to external constraints (e.g., the number of machines exceeds the budget; some services should not be merged because it requires reorganization of development teams). Instead of letting Elmo first explore the entire design space and then filtering the unacceptable solutions, SOA developers can specify additional model annotations upfront to constrain which refactorings Elmo will try to apply; corners of unacceptable solutions in the design space are skipped.

Specifically, in the initial deployment model, users can indicate that a service must remain intact (i.e., it cannot be split, merged, or modified); that some services cannot be merged; that some sets of types cannot be split; that a call must remain intact (i.e., it cannot be flipped or replaced by a queue/cache); that some sets of services must be collocated; that the number of machines must not exceed some limit.

6 Conclusion

Related work. Other tools exist that aid in refactoring existing architectures. These tools help to visualize architectures, detect code smells like dependency cycles, or validate architectural rules (e.g. [21,11,4,6]). However, these tools work at the implementation/code level and do not take deployment into account, nor can they evaluate performance sensitivities like Elmo does. Moreover, a key strength of Elmo is its rigorous foundation and formal correctness (i.e., the core contribution of this paper); these other tools do not provide such guarantees.

Application performance monitoring tools (e.g. [7,2,8,18]) provide a quick insight in interactions between services and aid in detecting performance problems. However, they can only do this when software is actually deployed; not during design. These tools can identify bottlenecks, but they have only very limited support for finding solutions. Based on the formalization presented in this paper, in contrast, Elmo can automatically compute series of refactorings.

UML component diagrams allow developers to document dependencies between components/services. A key difference with our approach is that component diagrams do not distinguish between pushes and pulls [14] (i.e., component diagrams model dependencies between components, but they do not model the direction and initiative of information flows that push and pull operations additionally convey); in our model, this is vital information to reason about refactorings and sensitivities. To provide such information in UML, complementary behavioral diagrams (e.g., UML sequence diagrams) can be used, but then the level of detail becomes too low for our purpose, while at the same time a maintenance burden emerges. Also, mixing different types of diagrams is cumbersome.

Future work. We are currently working along three axes: theory, implementation, and case studies. Along the theory axis, to better support situations where the specified insensitivity goals are inconsistent (i.e., impossible to achieve), we are developing notions of *Pareto efficiency* of deployment models. The idea is to devise formal machinery to compute *Pareto frontiers*: sets of deployment models such that no deployment model in the set can be be further refactored to eliminate an undesirable sensitivity without simultaneously (re)introducing one. We are also considering to incorporate a form of simulation to provide quantitative feedback on refactorings (e.g., [12]); this may be useful to analyze and reason about, for instance, latency (currently not supported).

Along the implementation axis, we are working on the optimizations stated in Sect. 5 (model annotations; partial order reduction; parallellization).

References

1. Akamai Technologies Inc: Akamai Online Retail Performance Report | Akamai (2017), Accessed 28 June 2019, `https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp`

2. AppDynamics LLC: Application Performance Monitoring & Management | AppDynamics (nd), Accessed 28 June 2019, https://www.appdynamics.com
3. Bertoli, M., Casale, G., Serazzi, G.: JMT: performance engineering tools for system modeling. SIGMETRICS Performance Evaluation Review **36**(4), 10–15 (2009)
4. Bischofberger, W.R., Kühl, J., Löffler, S.: Sotograph - A pragmatic approach to source code architecture conformance checking. In: EWSA. Lecture Notes in Computer Science, vol. 3047, pp. 1–9. Springer (2004)
5. Brebner, P.: Real-world performance modelling of enterprise service oriented architectures: delivering business value with complexity and constraints (abstracts only). SIGMETRICS Performance Evaluation Review **39**(3), 12 (2011)
6. Caracciolo, A., Lungu, M.F., Nierstrasz, O.: A unified approach to architecture conformance checking. In: WICSA. pp. 41–50. IEEE Computer Society (2015)
7. Datadog Inc: Modern monitoring & analytics | Datadog (nd), Accessed 28 June 2019, https://www.datadoghq.com
8. Dynatrace LLC: Software intelligence for the enterprise cloud | Dynatrace (nd), Accessed 28 June 2019, https://www.dynatrace.com
9. van Eekelen, M., Jongmans, S.S., Lamers, A.: Non-Quantitative Modeling of Service-Oriented Architectures, Refactorings, and Performance. Tech. Rep. TR-OU-INF-2017-02, Open University of the Netherlands (2017)
10. Elmo Demo (2018), Accessed 28 June 2019, https://youtu.be/0i9kxqh_GBs
11. Headway Software Technologies Ltd: Structure101 Home » Structure101 (nd), Accessed 28 June 2019, https://structure101.com
12. Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: A formal model of cloud-deployed software and its application to workflow processing. In: SoftCOM. pp. 1–6. IEEE (2017)
13. Juan Ferrer, A., et al.: OPTIMIS: A holistic approach to cloud service provisioning. Future Generation Comp. Syst. **28**(1), 66–77 (2012)
14. Kobryn, C.: Modeling components and frameworks with UML. Commun. ACM **43**(10), 31–38 (2000)
15. Kounev, S.: Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE Trans. Software Eng. **32**(7), 486–502 (2006)
16. Lamers, A., van Eekelen, M.C.J.D., Jongmans, S.: Improved architectures/deployments with elmo. In: ICSOC, Demonstration Track. Lecture Notes in Computer Science, vol. 11434, pp. 419–424. Springer (2018)
17. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
18. New Relic Inc: New Relic | Deliver more perfect software (nd), Accessed 28 June 2019, https://www.newrelic.com
19. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.M.: Microservices in practice, part 1: Reality check and service design. IEEE Software **34**(1), 91–98 (2017)
20. Peled, D.: Partial-order reduction. In: Handbook of Model Checking, pp. 173–190. Springer (2018)
21. SonarSource SA: Continuous Inspection | SonarQube (nd), Accessed 28 June 2019, https://www.sonarqube.org
22. Zhu, L., Liu, Y., Bui, N.B., Gorton, I.: Revel8or: Model driven capacity planning tool suite. In: ICSE. pp. 797–800. IEEE Computer Society (2007)