A Formal Web Services Architecture Model for Changing PUSH/PULL Data Transfer

Naoya Nitta[1], Shinji Kageyama[1], and Kouta Fujii[1]

Konan University, 8-9-1, Okamoto, Kobe, Japan
n-nitta@konan-u.ac.jp, m2124002@a.konan-u.ac.jp, m2224001@s.konan-u.ac.jp

Abstract. Deciding how data should be transferred among Web services is an important part of their architecture design. Basically, each piece of data is transferred in either PUSH or PULL style. The architect's selection of data transfer methods generally has a great impact on both the overall structure and performance of Web services. However, little work has been done on helping architects to select suitable data transfer methods. In this paper, we present a formal model to abstract some parts of Web services architecture that are not affected by the selection of data transfer methods, and based on the model, we propose an architecture level refactoring for changing data transfer methods. Also, we present an algorithm to generate prototypes of Web services from the architecture model and selected data transfer methods, and proved the correctness of the algorithm. Furthermore, we developed a tool that provides a graph-based UI for the refactoring and can generate executable prototypes of Web services. To evaluate our method, we conducted case studies for several Web applications and confirmed that the generated prototypes can be used to estimate the performance.

Keywords: Web services architecture model · PUSH/PULL data transfer · Architecture model refactoring.

1 Introduction

Many kinds of data are created, changed and transferred among Web services. Between two Web services, each piece of data is basically transferred in either PUSH or PULL style. In PUSH-style, the source side service sends data to the destination side one as a parameter of an API call, but in PULL-style, the destination side one gets data from the source side one as a response of an API call. Generally, selection of data transfer methods has a great impact on both the overall structure and performance of Web services. However, little work has been done on helping architects to select suitable data transfer methods.

In this paper, we present a formal architecture model to support selection of data transfer methods among Web services. Before explaining the model, first, we think about the meaning of selection of data transfer methods. For example, consider the case of transferring data from service A to service B. If PUSH-style has been selected for the transfer, then B's API is called by A and the control

flows in the same direction of the data-flow, but in PULL-style, A's API is called by B and the control flows in the reverse direction. Therefore, selection of a data transfer method corresponds to selection of the direction of the control-flow.

If some 'control-independent' parts of Web services architecture can be specified before selecting data transfer methods, then their selection and reselection tasks will be simplified. However, many formal architecture models [1, 6, 8] cannot be used for the purpose because they are basically process-centric and the process based descriptions are control-dependent. Here, by control-dependent, we mean that the model contains some information about control-flow and its descriptions are affected by a change of the structure of control-flow. Therefore in this paper, we introduce a formal architecture model in which control-dependent information is abstracted away. To make the model control-independent, each component of the model is assumed to have no temporal state and no internal transition. Therefore in the model, every state transition of every component is always observable and the states of all components are assumed to be changed synchronously. On the basis of the architecture model, selection of data transfer methods is regarded as addition of the information about the directions of control-flow to the model, and at least at the architecture level, the data transfer methods can be refactored simply by changing the added information. In this paper, we present an algorithm to generate a prototype of Web services from our architecture model and the added information, and prove the correctness of the algorithm. We also designed a simple architecture description language (ADL for short) to specify the architecture model, and developed a tool that can read a model file written in the ADL, provides a graph-based UI to select data transfer methods and can generate executable JAX-RS prototypes so that they are used to estimate the impact of selection of data transfer methods on the performance. To evaluate our method, we conducted case studies for several simple Web applications and confirmed that the generated prototypes can be used to estimate the performance.

2 Motivating Example

First, we show an example of Web services in which data transfer methods can be changed to PUSH or PULL-style. The subject services are simple weather observation services (in the following, WOS for short). For the simplicity, we assume that the services deal with only a single weather station. The functional requirements are as follows. Whenever the station observes the weather, the temperature in Fahrenheit is measured and that in Celsius is instantly calculated from the measured temperature. At the same time, the value of the highest temperature (in Fahrenheit) is updated using the measured temperature. In addition, the highest temperature is reset every time a date changes.

Here, assume that the services are planned to be constructed as RESTful [3] Web services and consist of three resources; the temperature in Fahrenheit (denoted by `temp_f`), the temperature in Celsius (denoted by `temp_c`) and the highest temperature (denoted by `highest`). We show the data-flow among the re-

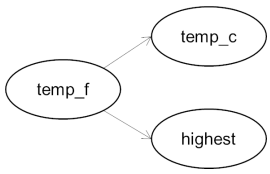
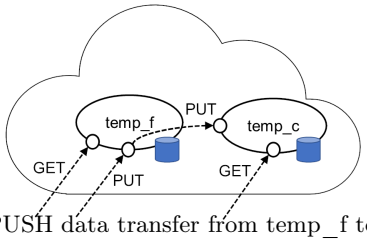
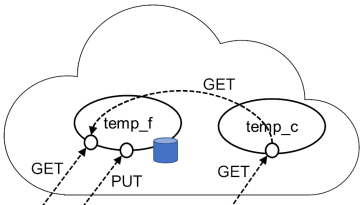


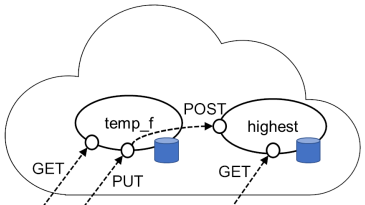
Fig. 1: Dataflow Graph of WOS System



(a) PUSH data transfer from temp_f to temp_c



(b) PULL data transfer from temp_f to temp_c



(c) PUSH data transfer from temp_f to highest

Fig. 2: Possible Implementations of Data Transfer in WOS System

sources in Fig. 1. In a RESTful style of architecture, each resource corresponds to a piece of information on the Web that can be identified by a URI. Each resource can provide HTTP methods such as GET, PUT, POST and DELETE. The GET method is assumed to be *safe*, which means that the state of the resource is not changed by applying the method, and the GET, PUT and DELETE methods are assumed to be *idempotent*, which means that its multiple applications to the resource cause the same effects on the resource as its single application. Typically, the GET method is used to obtain the state of a resource, and the PUT and POST methods are used to update the state of a resource.

First, consider to implement the data transfer between the `temp_f` and `temp_c` resources in PUSH-style (see Fig. 2(a)). For each resource, the latest state is stored within the resource and its GET method is implemented to return the state as its response. The method that is used to set a new state to `temp_f` is PUT since more than one application of the method does not change its state. Here, recall that the state of `temp_c` can be calculated from that of `temp_f`. Therefore in PUSH-style, whenever PUT method of `temp_f` is called, it also calls PUT method of `temp_c` so that its internal state is kept up-to-date.

Second, consider to implement the same data transfer in PULL-style (see Fig. 2(b)). As is the case with the PUSH-style, the latest state of `temp_f` is stored within the resource and it is returned by calling its GET method. However in PULL-style, the latest state of `temp_c` is not stored anywhere because it can be directly calculated from that of `temp_f`. Therefore, GET method of `temp_c` always calls GET method of `temp_f` to obtain its latest state. On the other hand, PUT method of `temp_f` does not call any other method.

Last, consider to implement the data transfer between the `temp_f` and the `highest` resources. Its PUSH-style implementation is similar to the above case (see Fig. 2(c)). However, the state of `highest` should be updated by POST method because every application of the method may change its state. The most remarkable thing about this data transfer is that it cannot be implemented safely in PULL-style. The reason is that the latest state of `highest` cannot be calculated only from that of `temp_f`. It should be recalculated whenever `temp_f` is updated and the previous state of `highest` is always needed in the recalculation. Therefore, `highest` should be notified of every update of `temp_f` and the latest state of `highest` should be stored within the resource. Note that periodic PULL-style data transfer to poll the latest state of `temp_f` is not a solution because not all updates of `temp_f` may be monitored. With respect to the daily reset operation of `highest`, the corresponding method can be implemented as PUT because the operation satisfies idempotency. By comparing the above implementations, we can make the following observations.

- **Some data transfer method cannot be changed to PULL-style.** In the example of WOS, the data transfer from `temp_f` to `highest` cannot be implemented safely in PULL-style, but that from `temp_f` to `temp_c` can be. Just looking at the data-flow in Fig. 1, we cannot distinguish these properties of data transfer. To determine the property, we need to know the concrete relation between the source and destination resources of the data transfer. Therefore, in the next section, we present a formal model that can represent the relation among state changes of the relevant resources.
- **Some resource's state should be stored within the resource and the other ones do not need to be stored.** At the implementation level, the state of a resource may not be stored within the resource even if it is always observable from the outside of the system and its value can always be obtained. For example, the state of `temp_c` is not stored anywhere in the PULL-style implementation although it is stored within the resource in the PUSH-style implementation. We can say that the state of a resource should be stored within the resource if it is updated by PUSH-style transfer.
- **Data transfer methods will affect the non-functional properties of the services.** Selection of the data transfer methods is tightly associated with the non-functional properties of the services. For example in a PUSH-style implementation, since the states of the destination resources should be stored in the memory or storage, it will increase the consumption. The communication load to update the resource's state is higher in PUSH-style than in PULL-style, but that to observe the resource's state is the opposite.

3 Overview of Architecture Model

In this paper, we present a formal architecture model in which control-dependent information is abstracted away. In this section, we explain its overview. Its formal definition is given in the next section. As stated in Sec. 1, process based descriptions are generally control-dependent. For example, consider the case that process P sends message b in response to receiving message a. In process-based models like CCS [7], P is modeled as having a halfway state where it has received a but still has not sent b. From the state, by performing an internal action, which is unobservable from the outside, P sends b and becomes the next state. We think such a halfway state and internal action make the computation model control-dependent. Therefore in our model, each component is assumed to have no halfway state and no internal transition. In addition, every state transition of every component is observable and the states of all components are changed synchronously. In our model, a *resource* is a software component c that satisfies:

1. c is identifiable from the outside of the system,
2. c seems to have its own state and the state is always observable from the outside of the system,
3. the state of c changes only when c receives or sends some message.

Note that these are just architecture level assumptions. By adding control-dependent information to the model, each component can be implemented to have halfway states and internal transitions.

We illustrate our architecture model based on the WOS explained in the previous section. In our architecture model, a resource is modeled as a state transition system whose state is changed by receiving or sending a message. In several existing architecture models (e.g. [5]), each component is modeled as a labeled transition system (LTS). However, since the state space of LTS is limited to finite, it does not have sufficient expressive power to represent many kinds of data structure such as natural numbers and lists. In addition, LTS is not suitable for generating a readable program since the state changes will be translated as a large number of conditional branches. Therefore in our model, we do not limit the state space of transition systems to finite. For example, we model the state transition of `temp_f` by calling its PUT method with parameter x as:

$$\mathrm{temp\_f}(f, x) = x.$$

Each state transition function takes a state of a resource and a message as its first and second arguments, and returns its next state. Similarly, the state transition functions to update `temp_c` and `highest` can be modeled as follows.

$$\mathrm{temp\_c}(c, x) = (x - 32)/1.8.$$
$$\mathrm{highest}(h, x) = \mathrm{if}(x > h, x, h).$$

In these definitions, we directly use the name of each resource as the name of its state transition function, and assume that these resources receive the same message x synchronously.

The above formulas may seem too concrete to be used in an architecture model. However, as discussed in the previous section, without knowing the concrete relationship between the source and destination resources, it is impossible to know whether the data transfer method can be changed to PULL-style or not. For example, consider why the data transfer from `temp_f` to `temp_c` can be implemented in PULL-style. In the above formulas, x corresponds to a new state of `temp_f`. As we can see from the formulas, the next state of `temp_c` (i.e., $(x - 32)/1.8$) can be calculated only from x, but to calculate the next state of `highest` (i.e., $\mathrm{if}(x > h, x, h)$), in addition to x, its current state h is also needed. This difference determines the implementability of data transfer in PULL-style. More generally, with respect to data transfer from resource s to resource d, if there exists a homomorphism that maps the state transition of s to that of d, then the data transfer can be implemented in PULL-style (see Sec. 5.3 for the detail). For example with respect to the data transfer from `temp_f` to `temp_c`, a function $h(x) = (x - 32)/1.8$ is a homomorphism since it satisfies:

$$h(\mathrm{temp\_f}(f, x)) = h(x) = (x - 32)/1.8 = \mathrm{temp\_c}(h(f), x).$$

By using h, the data transfer can be implemented in PULL-style. On the other hand, since there is no homomorphism from the state transition of `temp_f` to that of `highest`, it cannot be implemented in PULL-style. In this way, whether each data transfer can be implemented in PULL-style or not can be determined by comparing the state transition functions of the source and destination resources.

To make the source side and the destination side state transition functions comparable, in our architecture model, the formulas that define these functions are grouped by a *channel*. A channel is defined for each data transfer, and in each channel, we assume that the states of the source and destination resources are changed synchronously by the same message on the channel. Each channel is assumed to have three ports; the input, output and reference ports, and the source and destination resources are assumed to connect to its input and output ports, respectively. The state of a resource connected to the reference port does not change by the message on the channel. Semantically, these assumptions mean that a message is transferred from the input side and reference side resources to an output side one through the channel instantaneously, and the state change on the input side resource, the data transfer and the state change on the output side resource are all synchronized. Such assumptions make the architecture model capable of representing data-flow but independent of control-flow. Here, assume that data from `temp_f` to `temp_c` is transferred through a channel named c_1. Then, in our architecture model, the data transfer can be represented as:

$$\begin{aligned} \mathrm{temp\_f}^{c_1,\mathrm{I}}(f, y) &= y, \\ \mathrm{temp\_c}^{c_1,\mathrm{O}}(c, y) &= (y - 32)/1.8. \end{aligned}$$

In these formulas, the superscript of each function consists of the channel and its port I, O or R that the resource connects to. In this data transfer, y is the transferred message from `temp_f` to `temp_c` on channel c_1 and the message is calculated from the next state of `temp_f`, which is the input side resource of c_1.

Table 1: Connection between Channels and Resources in WOS

Channel	Input side resource	Output side resource
$c_{\mathrm{I/O}_1}$		temp_f
$c_{\mathrm{I/O}_2}$		highest
c_1	temp_f	temp_c
c_2	temp_f	highest

In general, multiple resources can connect to the input and reference ports of a channel, and a message on the channel is calculated from the current and next states of the input side resources and the current states of the reference side resources. For example, consider the following channel c.

$$\begin{aligned}\mathrm{pos\_x}^{c,\mathrm{I}}(x, \langle dx, dy\rangle) &= x + dx = x',\\ \mathrm{pos\_y}^{c,\mathrm{I}}(y, \langle dx, dy\rangle) &= y + dy = y',\\ \mathrm{distance}^{c,\mathrm{O}}(d, \langle dx, dy\rangle) &= d + \sqrt{dx^2 + dy^2} = d'.\end{aligned}$$

The input side resources `pos_x` and `pos_y` represent the 2D position of someting and the output side resource `distance` repesents its travel distance. In this case, the message $\langle dx, dy\rangle$ is calculated from x, x', y and y' as $\langle x' - x, y' - y\rangle$. However, such a calculation may seem counterintuitive because the constraints that the calculation requires are a kind of inverse mapping of the input side state transition functions but the given constraints are in the form of state transition functions themselves. The reasons why we define such constraints for each channel separately and in the form of state transition functions are as follows.

- Since the massages sent on each channel generally differ depending on the channel, the input side constraints used to calculate the messages also differ depending on the channel, and should be defined separately.
- As stated before, to determine the changeability of the data transfer method to PULL-style, both the input side and output side constraints should be given in the state transition function form.
- When multiple resources connect to input and/or reference ports, as an inverse mapping, the constraints must be given in a multi valued inverse mapping for each resource (e.g., $x, x' \mapsto \langle x' - x, dy\rangle$ for `pos_x`), or a single valued inverse mapping of all input/reference side resources (e.g., $x, x', y, y' \mapsto \langle x' - x, y' - y\rangle$). However, both definitions also seem counterintuitive.
- If both the input side and output side constraints have the same form, then flipping their sides can flexibly be done.

The interactions between the system and its environment except for the observation of each resource's state are done through special channels for I/O. The observation of the state is assumed to be done directly from the environment without going through any channel.

In the following, we explain how the whole WOS can be modeled as an architecture model. In this example, we use channels $c_{\mathrm{I/O}_1}, c_{\mathrm{I/O}_2}, c_1$ and c_2, and let

$\mathrm{temp\_f}^{c_{\mathrm{I/O}_1},\mathrm{O}}(f,x) = x,$
$\mathrm{highest}^{c_{\mathrm{I/O}_2},\mathrm{O}}(h,v) = v,$
$\mathrm{temp\_f}^{c_1,\mathrm{I}}(f,y) = y,$ $\mathrm{temp\_c}^{c_1,\mathrm{O}}(c,y) = (y-32)/1.8,$
$\mathrm{temp\_f}^{c_2,\mathrm{I}}(f,z) = z,$ $\mathrm{highest}^{c_2,\mathrm{O}}(h,z) = \mathrm{if}(z > h, z, h).$

Fig. 3: Data Transfer Architecture Model of WOS

$c_{\mathrm{I/O}_1}$ and $c_{\mathrm{I/O}_2}$ be I/O channels. We show the connection between the resources and the channels in Table 1. The state transition functions of the resources are defined in Fig. 3. In our model, the state transition function of the same resource seems to be multiply defined in different channels, but they are just different images of the same function. Therefore, the first arguments of them and their function values are always identical, respectively. For example in Fig. 3, always $x = y = z$ holds. The whole system reacts in response to a message from the environment through an I/O channel. For example, if message 68 is input to the WOS through channel $c_{\mathrm{I/O}_1}$, then $\texttt{temp\_f}^{c_{\mathrm{I/O}_1},\mathrm{O}}(f, 68)$ is evaluated and the next state of `temp_f` is shown to be 68. Since $\texttt{temp\_f}^{c_{\mathrm{I/O}_1},\mathrm{O}}$, $\texttt{temp\_f}^{c_1,\mathrm{I}}$ and $\texttt{temp\_f}^{c_2,\mathrm{I}}$ are dirrefent images of the same function, we have $x = y = z = 68$. By these equations, both messages sent on c_1 and c_2 become 68. Finally, the states of `temp_c` and `highest` respectively become 20 and 68 if the previous state of `highest` is less than 68.

4 Data Transfer Architecture Model

4.1 Basic Definitions

We define a data transfer architecture model as $\mathcal{R} = \langle R, C, \rho, D, \tau, \mu, \Delta, s_0 \rangle$ where:

- R: a finite set of *resources*,
- C: a finite set of *channels*,
- ρ ($\rho : C \times \{\mathrm{I}, \mathrm{O}, \mathrm{R}\} \to 2^R$): a *connection map* that maps a channel and its port (input, output or reference) to the set of all connecting resources,
- D: a finite/infinite set of data values,
- τ ($\tau : R \to 2^D$): a map from a resource to the set of its all states,
- μ ($\mu : C \to 2^D$): a map from a channel to the set of all messages that can be transferred on the channel,
- $\Delta = \langle \delta_r^{c,d} \rangle_{c \in C, d \in \{\mathrm{I,O,R}\}, r \in \rho(c,d)}$: a finite set of state transition functions,
- s_0: the initial state ($s_0(r) \in \tau(r)$ for each $r \in R$).

The details of the model will be explained in the following.

4.2 Resource and Channel

A resource is a component that seems to have its own state and the state is always observable from the outside of the system. The set of all resources within the system is denoted by R. In this paper, we assume that the number of all resources is fixed and no resource is either created or deleted at runtime. However, each resource can have an infinite state space. A channel is used to synchronize the state changes in relevant resources. The set of all channels in the system is denoted by C. Each channel in C can have one input, one output and one reference ports, and to each port, an arbitrary number of resources can connect. For each channel $c \in C$, $\rho(c, \mathrm{I})$, $\rho(c, \mathrm{O})$ and $\rho(c, \mathrm{R})$ represent the sets of all resources that connect to the input, output and reference ports of c, respectively. These sets are assumed to be disjoint. For simplicity, we call $r \in \rho(c, \mathrm{I})$, $r' \in \rho(c, \mathrm{O})$ and $r'' \in \rho(c, \mathrm{R})$ an input side, output side and reference side resources of c, respectively. If the state of an input side resource of a channel c is changed, then a message m is sent on c and the state of an output side resource of c is changed by receiving m. C always contains non-empty set $C_{\mathrm{I/O}}$. Every I/O channel $c_{\mathrm{I/O}} \in C_{\mathrm{I/O}}$ has no input side resource, that is, $\rho(c_{\mathrm{I/O}}, \mathrm{I}) = \emptyset$.

4.3 States and Messages

In our architecture model, a data value is either a state of a resource or a message on a channel. The set of all data values is denoted by D. For each resource $r \in R$, $\tau(r)$ represents the set of all possible states of r, and for each channel $c \in C$, $\mu(c)$ represents the set of all possible messages that can be transferred on c. For each $c \in C$, an identity element $e_c \in \mu(c)$ that represents no operation is defined.

4.4 State Transition Functions

Δ is a finite set of state transition functions. It contains exactly one function for each connection of a resource to a channel. If a resource r connects to d side of a channel c (that is, $r \in \rho(c, d)$), then we denote the corresponding state transition function by $\delta_r^{c,d} : \tau(r) \times \mu(c) \to \tau(r)$. Each function takes a previous state of r and a message on c as the first and second arguments, and returns the next state of r. For each $c \in C$ and $r \in R$, $\delta_r^{c,\mathrm{O}}$ is always total and single valued, $\delta_r^{c,\mathrm{I}}$ may be partial and/or multi valued, and $\delta_r^{c,\mathrm{R}}$ is always multi valued and may be partial. Especially for $\delta_r^{c,\mathrm{I}}$ and $\delta_r^{c,\mathrm{R}}$, a more explanation would be needed. If $\delta_r^{c,\mathrm{R}}(x, y) = z$ holds, then the value of y is always uniquely determined from the value of x regardless of the value of z, and thus $\delta_r^{c,\mathrm{R}}$ is multi valued and may be partial. On the other hand, if $\delta_r^{c,\mathrm{I}}(x, y) = z$ holds, then there can be some constraint on y and z such that the value of y is uniquely determined from the value of z, and thus $\delta_r^{c,\mathrm{I}}$ may be multi valued and may be partial. For example, consider $\delta_{\mathrm{history}}^{c,\mathrm{I}}(h, \max(h2)) = h2$ where `history` is a resource whose state is a list of something and $\max(x)$ is the maximum value of the elements in list x. In this case, $h2$ is not uniquely determined from h and $\max(h2)$, and $\delta_{\mathrm{history}}^{c,\mathrm{I}}$ is multi valued. For a channel $c \in C$, let $\{r_1, \ldots, r_l\} = \rho(c, \mathrm{I})$, and $\{r'_1, \ldots, r'_k\} = \rho(c, \mathrm{R})$.

Then, to guarantee that a message on c can always be calculated from the current and next states of input side resources and the current states of reference side resources, there must be a *message generation function* $\delta^{-1,c}$ that satisfies for any $s_{r_1} \in \tau(r_1), \ldots, s_{r_l} \in \tau(r_l), s_{r'_1} \in \tau(r'_1), \ldots, s_{r'_k} \in \tau(r'_k)$,

$$m = \delta^{-1,c}(s_{r_1}, \delta^{c,\mathrm{I}}_{r_1}(s_{r_1}, m), \ldots, s_{r_l}, \delta^{c,\mathrm{I}}_{r_l}(s_{r_l}, m), s_{r'_1}, \ldots, s_{r'_k}). \tag{1}$$

4.5 Dataflow Graph and Validity of Architecture Model

For a given data transfer architecture model $\mathcal{R}$, a *dataflow graph* is a directed graph $G_{\mathcal{R}} = (N_R, N_C, E)$ such that $N_R = R$, $N_C = C$ and $E = \{\langle c, r\rangle \mid c \in C, r \in \rho(c, \mathrm{O})\} \cup \{\langle r, c\rangle \mid c \in C, r \in \rho(c, \mathrm{I})\}$. For each input channel $c \in C_{\mathrm{I/O}}$, a *dynamic dataflow graph* is the maximum subgraph $G_{\mathcal{R},c}$ of $G_{\mathcal{R}}$ such that every node in $G_{\mathcal{R},c}$ is reachable from node $c \in N_C$. We say $\mathcal{R}$ is *valid* when all of the following conditions are satisfied.

- $G_{\mathcal{R}}$ has no strongly connected component.
- For any input channel $c \in C_{\mathrm{I/O}}$, $G_{\mathcal{R},c}$ is a directed tree.
- For any channel $c \in C$ and $r \in \rho(c, \mathrm{R})$, if r is contained in $G_{\mathcal{R},c_{\mathrm{I/O}}}$, then r is a descendant of c on $G_{\mathcal{R},c_{\mathrm{I/O}}}$.

4.6 State Transition of the Whole System

A state of the whole system is a composition of states of all resources. We model a state s of the whole system as a map from the resources to their states. That is, for each resource $r \in R$, s satisfies $s(r) \in \tau(r)$. Also the initial state s_0 of the system satisfies the same condition.

The whole system reacts in response to a message from the environment through an I/O channel. More specifically, if the state of $\mathcal{R}$ changes from s to s' by receiving a message m on $c_{\mathrm{I/O}} \in C_{\mathrm{I/O}}$, then we write $s \overset{\langle m, c_{\mathrm{I/O}}\rangle}{\underset{\mathcal{R}}{\rightarrow}} s'$, and the state change is possible if and only if there exists a *message assignment* $\pi : C \to D$ such that:

- for every $c \in C$, $\pi(c) \in \mu(c)$,
- $\pi(c_{\mathrm{I/O}}) = m$,
- for each $c \in C$, if c is contained in $G_{\mathcal{R},c_{\mathrm{I/O}}}$, then for $d \in \{\mathrm{I}, \mathrm{O}, \mathrm{R}\}$ and $r \in \rho(c, d)$,

$$\delta^{c,d}_r(s(r), \pi(c)) \begin{cases} = s'(r) & (\delta^{c,d}_r \text{ is single valued}) \\ \ni s'(r) & (\delta^{c,d}_r \text{ is multi valued}), \end{cases} \tag{2}$$

- for each $c \in C$, if c is not contained in $G_{\mathcal{R},c_{\mathrm{I/O}}}$, then $\pi(c) = e_c$ and $s(r) = s'(r)$ for each r such that:
  - $r \in \rho(c, \mathrm{I})$, or
  - $r \in \rho(c, \mathrm{O})$ and r is not contained in $G_{\mathcal{R},c_{\mathrm{I/O}}}$.

For example, with respect to the architecture model explained in Sec. 3, $\{\text{temp\_f} \mapsto 66.2, \text{temp\_c} \mapsto 19, \text{highest} \mapsto 67.1\} \xrightarrow[\mathcal{R}]{\langle 68, c_{\mathrm{I/O}_1}\rangle} \{\text{temp\_f} \mapsto 68, \text{temp\_c} \mapsto 20, \text{highest} \mapsto 68\}$ holds since $c_{\mathrm{I/O}_2}$ is not contained in $G_{\mathcal{R},c_{\mathrm{I/O}_1}}$ and there exists a message assignment $\pi = \{c_{\mathrm{I/O}_1} \mapsto 68, c_{\mathrm{I/O}_2} \mapsto e_{c_{\mathrm{I/O}_2}}, c_1 \mapsto 68, c_2 \mapsto 68\}$. In the following, for an input sequence $\sigma = \langle m_1, c_1\rangle \cdots \langle m_n, c_n\rangle$, we write a concatenation of transition relations $\xrightarrow[\mathcal{R}]{\langle m_1,c_1\rangle} \cdots \xrightarrow[\mathcal{R}]{\langle m_n,c_n\rangle}$ as $\xRightarrow[\mathcal{R}]{\sigma}$.

5 RESTful Web Services Generation from Architecture Model and Selected Data Transfer Methods

As stated in Sec. 3, our architecture model is designed to be control-independent and not to be affected by changes of data transfer methods. Therefore, based on the architecture model, selection of data transfer methods is regarded as addition of the information about control-flow to the model. By adding such information, a prototype of RESTful Web services can be generated. In this section, we show a method to generate executable Web services from a data transfer architecture model and data transfer methods selected by a user. The generated prototype may be used to estimate the impact of selection of data transfer methods on the services performance.

5.1 Common Structure of RESTful Web Services for PUSH and PULL Data Transfer

As RESTful Web services, we consider to generate an executable prototype of JAX-RS Web application. JAX-RS is a Java API for RESTful Web services. Generally, a JAX-RS application consists of multiple resource classes, which are Java classes with some annotations. Let $\mathcal{R} = \langle R, C, \rho, D, \tau, \mu, \Delta, s_0\rangle$ be an arbitrary data transfer architecture model. Since $\mathcal{R}$ has no information about control-flow, selection of data transfer methods is required to generate a JAX-RS prototype. Let E^{PLL} be the set of all data transfer methods where PULL-style are selected, that is, $E^{\mathrm{PLL}} \stackrel{\text{def}}{=} \{\langle r, c\rangle \mid r \in R, c \in C, r \in \rho(c, \mathrm{I}),$ and the data transfer method between r and c is PULL-style$\}$. Basically, from $\mathcal{R}$, a JAX-RS prototype $\mathcal{P}_{\mathcal{R}}$ is generated by mapping R to resource classes. For each resource $r \in R$, let P_r be the resource class corresponding to r. We define $\mathrm{In_C}(r) \stackrel{\text{def}}{=} \{c \in C \mid r \in \rho(c, \mathrm{O})\}$ and $\mathrm{Out_C}(r) \stackrel{\text{def}}{=} \{c \in C \mid r \in \rho(c, \mathrm{I})\}$. Also for any subset $C' \subseteq C$ of channels, we define $\mathrm{In_R}(C') \stackrel{\text{def}}{=} \bigcup_{c\in C'} \rho(c, \mathrm{I})$ and $\mathrm{Out_R}(C') \stackrel{\text{def}}{=} \bigcup_{c\in C'} \rho(c, \mathrm{O})$. Then, for any resource $r \in R$, P_r always has one getter (GET), $|\mathrm{In_C}(r) \cap C_{\mathrm{I/O}}|$ input (PUT/POST), and at most $|\mathrm{In_R}(\mathrm{In_C}(r) \setminus C_{\mathrm{I/O}})|$ update (PUT/POST) methods.

5.2 Generation of PUSH-first RESTful Web Services

Let $\mathcal{R}$ be an arbitrary data transfer architecture model. For any data transfer method in $\mathcal{R}$, we can always select PUSH-style. The JAX-RS prototype that is

generated by selecting PUSH-style for every data transfer method (i.e., $E^{\mathrm{PLL}} = \emptyset$) is called *PUSH-first prototype* and written as $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$. In this subsection, we will explain how to generate $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$ from $\mathcal{R}$.

First, for each resource $r \in R$, P_r always has a *state fileld*, that is a field to store its own state. The getter method of P_r is defined as follows.

```
public TypeOf_r get() {
   return this.stateOf_r;   // the state field of r
}
```

Next, let $\{r_1, \ldots, r_l\} = \mathrm{In_R}(\mathrm{In_C}(r))$. Then, for each j $(1 \leq j \leq l)$, P_r has a *cache field* of r_j, that is a field to store the cache of the latest state of r_j.

Last, we define the update and input methods of P_r. Let $\{r'_1, \ldots, r'_k\} = \mathrm{Out_R}(\mathrm{Out_C}(r))$ and consider each channel $c \in \mathrm{In_C}(r)$. If $c \notin C_{\mathrm{I/O}}$, then $|\mathrm{In_R}(\{c\})|$ update methods are defined for c. More concretely, for each $r_j \in \mathrm{In_R}(\{c\})$ (here, j satisfies $1 \leq j \leq l$), an update method of P_r for r_j is defined as follows.

```
public void update_from_r_j(TypeOf_r_j r_j) {
   this.stateOf_r = δ_r^{c,O}(this.stateOf_r,
         δ^{-1,c}(this.cacheOf_r_1, this.cacheOf_r_1,...,this.cacheOf_r_j, r_j,...));
   this.cacheOf_r_j = r_j;
   r'_1.update_from_r(this.stateOf_r);
      ⋮
   r'_k.update_from_r(this.stateOf_r);
}
```

If $c \in C_{\mathrm{I/O}}$, then the input method for c is defined as follows.

```
public void input_on_c(TypeOf_m m) {
   this.stateOf_r = δ_r^{c,O}(this.stateOf_r, m);
   r'_1.update_from_r(this.stateOf_r);
      ⋮
   r'_k.update_from_r(this.stateOf_r);
}
```

5.3 Generation of PULL-containing Web Services

In this subsection, we consider to change each data transfer method in $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$ to PULL-style. Let r be a resource. For convenience of explanation, we assume that $\mathrm{Out_C}(r) = \{c'\}$ for some $c' \in C$ and let $\{r'_1, \ldots, r'_k\} = \mathrm{Out_R}(\{c'\})$. If $\langle r, c' \rangle \in E^{\mathrm{PLL}}$, then the generated prototype should satisfy the following conditions.

First, each resource class $P_{r'_i}$ $(1 \leq i \leq k)$ has no state field and no update method. The getter method of $P_{r'_i}$ is redefined as:

```
public TypeOf_r'_i get() {
   return f_{r'_i}(..., r.get(), ...);
}
```

where $f_{r'_i}$ is a homomorphism that calculates the latest state of r'_i and will be explained below. $P_{r'_i}$ has no cache field of r.

Next, each update method of P_r is redefined as follows.

```
public void update_from_r_j(TypeOf_r_j r_j) {
```

```
    this.stateOf_r = δ_r^{c,O}(this.stateOf_r,
          δ^{-1,c}(this.cacheOf_r_1, this.cacheOf_r_1,...,this.cacheOf_r_j, r_j,...));
    this.cacheOf_r_j = r_j;
}
```

By changing more than one data transfer method in $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$ to PULL-style as the above, we can generate a *PULL-containing prototype* $\mathcal{P}_{\mathcal{R}}$. However, as discussed in Sec. 2, not all data transfer methods can be safely changed to PULL-style. In the following, we call the prototype in which as many data transfer methods as possible are changed to PULL-style *PULL-first*. In the next section, we will prove that $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$ and any PULL-containing prototype $\mathcal{P}_{\mathcal{R}}$ are equivalent if $\mathcal{P}_{\mathcal{R}}$ and $\mathcal{R}$ satisfy the following conditions.

Condition 1 *For every resource $r \in R$ and channel $c \in \mathrm{Out_C}(r)$, if $\langle r, c\rangle \in E^{\mathrm{PLL}}$, then for every resource $r' \in \mathrm{Out_R}(\{c\})$ and channel $c' \in \mathrm{Out_C}(r')$,*

1. $\langle r', c'\rangle \in E^{\mathrm{PLL}}$,
2. $\mathrm{In_C}(r') = \{c\}$, *and*
3. $\rho(c, \mathrm{R}) = \emptyset$ *holds.*

Condition 2 *For every channel $c \in C$, let $\{r_1, \ldots, r_l\} = \mathrm{In_R}(\{c\})$ and if $\langle r_j, c\rangle \in E^{\mathrm{PLL}}$ for some j $(1 \le j \le l)$, then for each resource $r' \in \mathrm{Out_R}(\{c\})$,*

$$s_0(r') = f_{r'}(s_0(r_1), \ldots, s_0(r_l)), \tag{3}$$

and for any state s of $\mathcal{R}$ and any message $m \in \mu(c)$ on c,

$$\delta_{r'}^{c,\mathrm{O}}(f_{r'}(s(r_1), \ldots, s(r_l)), m) = f_{r'}(\delta_{r_1}^{c,\mathrm{I}}(s(r_1), m), \ldots, \delta_{r_l}^{c,\mathrm{I}}(s(r_l), m)). \tag{4}$$

This means that there exists a homomorphism $f_{r'}$ from the state transition systems of the source resources of c to that of each destination resource of c.

6 Equivalence of RESTful Web Services and Data Transfer Architecture Model

In this section, we give a brief outline of the proofs of the equivalence between a data transfer architecture model and any JAX-RS prototype generated from the model (for the details of the proofs, see the online appendix[1]). In advance of the proofs, first, we define some notations. Let $\mathcal{R}$ be an arbitrary data transfer architecture model and $\mathcal{P}_{\mathcal{R}}$ be a JAX-RS prototype generated from $\mathcal{R}$. For $\mathcal{P}_{\mathcal{R}}$, a *communication* $m(r, x_1, \ldots, x_n)/v$ is a pair of a request $m(r, x_1, \ldots, x_n)$ and its response v where m is a method of a resource r and $x_1, \ldots, x_n$ are parameters. Also for $\mathcal{P}_{\mathcal{R}}$, we consider a communication sequence $\widehat{\langle m, c_{\mathrm{I/O}}\rangle}$ that corresponds to a message $\langle m, c_{\mathrm{I/O}}\rangle$ for $\mathcal{R}$. Let $\{r_1, \ldots, r_k\} = \rho(c_{\mathrm{I/O}}, \mathrm{O})$. Then,

$$\widehat{\langle m, c_{\mathrm{I/O}}\rangle} \stackrel{\mathrm{def}}{=} \mathrm{input\_on\_}c_{\mathrm{I/O}}(r_1, m)/\mathrm{void} \cdots \mathrm{input\_on\_}c_{\mathrm{I/O}}(r_k, m)/\mathrm{void}.$$

[1] https://nitta-lab.github.io/appendix_FACS2023

In addition, let $\widehat{\sigma} = \widehat{\langle m_1, c_1\rangle} \cdots \widehat{\langle m_n, c_n\rangle}$ if $\sigma = \langle m_1, c_1\rangle \cdots \langle m_n, c_n\rangle$ is an input sequence for $\mathcal{R}$. For any state s of $\mathcal{R}$, we write a state of $\mathcal{P}_\mathcal{R}$ as $\mathcal{P}_\mathcal{R}(s)$ if for each resource class P_r of a resource r, $s(r)$ is stored in the state field in P_r and $s(r')$ is stored in any cache field of r' in P_r. If the state of $\mathcal{P}_\mathcal{R}$ can be changed from $\mathcal{P}_\mathcal{R}(s)$ to $\mathcal{P}_\mathcal{R}(s')$ by a communication $m(r, x_1, \ldots, x_n)/v$, then we write $\mathcal{P}_\mathcal{R}(s) \overset{m(r,x_1,\ldots,x_n)/v}{\longrightarrow} \mathcal{P}_\mathcal{R}(s')$. For a communication sequence $\omega = \alpha_1 \cdots \alpha_n$, we write a concatenation of transition relations $\overset{\alpha_1}{\to} \cdots \overset{\alpha_n}{\to}$ as $\overset{\omega}{\Rightarrow}$.

Here, we prove the equivalence between a data transfer architecture model and the generated PUSH-first prototype.

Theorem 1. *Let $\mathcal{R} = \langle R, C, \rho, D, \tau, \mu, \Delta, s_0\rangle$ be an arbitrary valid data transfer architecture model. Then, for any input sequence σ, $\mathcal{P}_\mathcal{R}^{\mathrm{PSH}}(s_0) \overset{\widehat{\sigma}}{\Longrightarrow} \mathcal{P}_\mathcal{R}^{\mathrm{PSH}}(s)$ iff $s_0 \overset{\sigma}{\underset{\mathcal{R}}{\Longrightarrow}} s$.*

Proof Sketch. The theorem is proved by the definition of $\mathcal{P}_\mathcal{R}^{\mathrm{PSH}}$ and induction on the number $n = |R|$ of the resources. For the induction step, $R' = R \setminus \{r\}$ (where $\mathrm{Out_C}(r) = \emptyset$) is selected, and that the value stored in the state field of the resource class of r always equals to $s(r)$ is shown. □

Next, we prove the equivalence between the PUSH-first prototype and any PULL-containing prototype generated from the same model.

Theorem 2. *Let $\mathcal{R} = \langle R, C, \rho, D, \tau, \mu, \Delta, s_0\rangle$ be an arbitrary valid data transfer architecture model, and $\mathcal{P}_\mathcal{R}$ be any JAX-RS prototype generated from $\mathcal{R}$ and satisfying conditions 1 and 2. Then, $\mathcal{P}_\mathcal{R}^{\mathrm{PSH}}$ and $\mathcal{P}_\mathcal{R}$ satisfy*

$$\begin{aligned}\forall_\sigma.\forall_{r\in R}.\ \{s_r \mid \exists_s.\mathcal{P}_\mathcal{R}^{\mathrm{PSH}}(s_0) \overset{\sigma}{\Rightarrow} \mathcal{P}_\mathcal{R}^{\mathrm{PSH}}(s) \overset{\mathrm{get}(r)/s_r}{\Longrightarrow} \mathcal{P}_\mathcal{R}^{\mathrm{PSH}}(s)\}\\ = \{s'_r \mid \exists_{s'}.\mathcal{P}_\mathcal{R}(s_0) \overset{\sigma}{\Rightarrow} \mathcal{P}_\mathcal{R}(s') \overset{\mathrm{get}(r)/s'_r}{\Longrightarrow} \mathcal{P}_\mathcal{R}(s')\}.\end{aligned}$$

Proof Sketch. The theorem is proved by double induction on the length of σ and the number $n = |R|$ of the resources. For the induction on n, also $R' = R \setminus \{r\}$ (where $\mathrm{Out_C}(r) = \emptyset$) is selected, and that the responses of the getter methods of r in $\mathcal{P}_\mathcal{R}^{\mathrm{PSH}}$ and that in $\mathcal{P}_\mathcal{R}$ are always equal is shown. □

Last, we prove the equivalence between a data transfer architecture model and any generated JAX-RS prototype.

Theorem 3. *Let $\mathcal{R} = \langle R, C, \rho, D, \tau, \mu, \Delta, s_0\rangle$ be an arbitrary valid data transfer architecture model, and $\mathcal{P}_\mathcal{R}$ be any JAX-RS prototype generated from $\mathcal{R}$ and satisfying conditions 1 and 2. For an arbitrary input sequence σ, $s_0 \overset{\sigma}{\underset{\mathcal{R}}{\Longrightarrow}} s$ holds if and only if there exists a state s' of $\mathcal{P}_\mathcal{R}$ such that $\mathcal{P}_\mathcal{R}(s_0) \overset{\widehat{\sigma}}{\Rightarrow} \mathcal{P}_\mathcal{R}(s') \overset{\mathrm{get}(r)/s(r)}{\Longrightarrow} \mathcal{P}_\mathcal{R}(s')$ holds for any resource $r \in R$.*

Proof. The theorem follows from theorems 1 and 2. □

7 Architecture Level Refactoring for Changing Data Transfer Methods

As stated in the previous section, from any valid data transfer architecture model $\mathcal{R}$, the PUSH-first JAX-RS prototype $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$ can be directly generated. Also, any PULL-containing JAX-RS prototype $\mathcal{P}_{\mathcal{R}}$, in which more than one data transfer method is changed to PULL-style can be generated from $\mathcal{P}_{\mathcal{R}}^{\mathrm{PSH}}$. In this section, as an architecture level refactoring, we consider regenerating the PUSH-first or a PULL-containing JAX-RS prototype directly from $\mathcal{R}$. In the refactoring process, a user is required to select one combination of data transfer methods from all safe ones that satisfy the conditions shown in Sec. 5.3. The whole process of the refactoring is summarized as follows.

1. Check the validity of $\mathcal{R}$.
2. For each data transfer, PULL-style is determined to be selectable if the conclusion parts of condition 1-(2), condition 1-(3) and condition 2 are satisfied, and every data transfer method is initialized to PUSH-style.
3. Ask the user if he/she wants to change each data transfer method to PULL-style if possible. Throughout the selection process, condition 1-(1) is always checked, and the selection that violates the condition is not asked.
4. Generate a JAX-RS prototype based on the algorithm shown in Sec. 5.
5. Back to step 3 if the user wants to reselect other data transfer methods.

Among the above steps, only the check of the conclusion part of condition 2 in step 2 is considered difficult because determining the existence of a homomorphism is generally hard. Therefore in this paper, we consider to check a sufficient condition, *right unaryness* of the input and output side state transition functions. A function f is called right unary if f satisfies

$$f(x, z) = f(y, z) \tag{5}$$

for any x and y. The sufficiency of the right unaryness can be shown as follows. Let c be a channel, and assume that $\{r_1, \ldots, r_l\} = \rho(c, \mathrm{I})$ and $r' \in \rho(c, \mathrm{O})$. If for every i $(1 \leq i \leq l)$, $\delta_{r_i}^{c,\mathrm{I}}$ is right unary, then there exists a function δ^{-1} such that:

$$m = \delta^{-1}(\delta_{r_1}^{c,\mathrm{I}}(s_1, m), \ldots, \delta_{r_l}^{c,\mathrm{I}}(s_l, m)),$$

and if $\delta_{r'}^{c,\mathrm{O}}$ is also right unary, then a function $f_{r'}$ that satisfies

$$f_{r'}(s'_1, \ldots, s'_l) = \delta_{r'}^{c,\mathrm{O}}(z, \delta^{-1}(s'_1, \ldots, s'_l))$$

for any constant z becomes homomorphic and satisfies equation (4).

8 Tool Implementation

We defined a simple architecture description language (ADL for short) based on *many-sorted algebra* [4], and developed a graph-based refactoring tool[2] shown in

[2] https://github.com/nitta-lab/DataTransferModelingTool

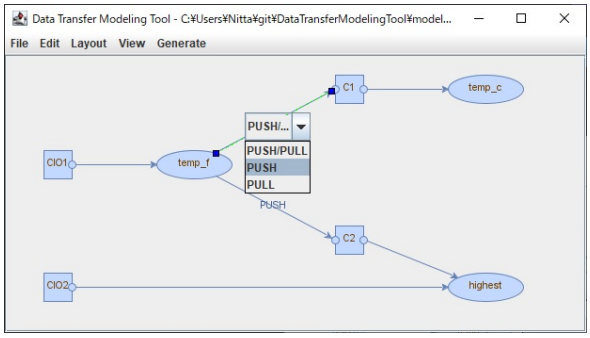


Fig. 4: Graph-based Tool To Refactor Data Transfer Methods

Fig. 4. The tool first reads a model file written in the ADL and performs the steps 1 and 2 of the process shown in the previous section. Then, a data-flow graph is displayed on the tool and the user is allowed to select a safe combination of data transfer methods through pull-down menus. Last, the tool generates an executable JAX-RS prototype. In JAX-RS prototype generation, if the state transition function of a resource is right unary, then an API to update the resource state is implemented as PUT method otherwise POST method.

9 Case Studies and Discussion

9.1 Case Studies

As case studies, we designed the architectures of the following three applications and wrote the model files in the ADL.

- **WOS (Weather Observation Services)**: As a case study, we used the WOS explained in Sec. 3.
- **Inventory Management Services**: As another case study, we wrote a model file for inventory management services for a liquor store. The services take receiving reports and shipping requests as input and keep track of warehouse inventory and the waiting list.
- **Online Card Game**: The last one is an online game based on Algo. In Algo, each player is dealt a certain number of uniquely numbered cards. The cards are laid face-down and each player is required to guess the number of another player's card in turn. The player who is guessed all the cards loses the game.

We generated the PULL-first/PUSH-first JAX-RS prototypes. The number of resources and the total lines of the model files and generated prototypes are summarized in Table 2. To confirm the effectiveness of the generated prototypes for performance estimation, we measured the computation time of each API call on the generated PULL-first/PUSH-first prototypes of the WOS. For the experiments, we deployed each JAX-RS prototype and a client program on the

Table 2: Total lines of model files and generated JAX-RS prototypes

Application	# of resources	Model File	PUSH-first	PULL-first
WOS	3	14	77	73
Inventory Management	7	43	361	346
Online Card Game	15	113	635	577

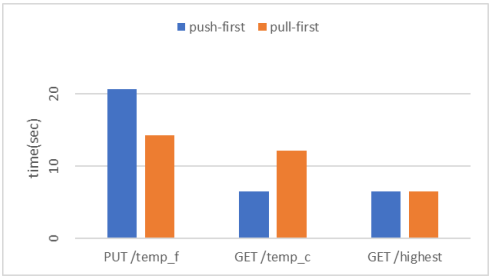


Fig. 5: Computation Time on JAX-RS Prototypes of WOS

same PC[3], and measured the time to perform 10,000 iterations of API call five times and calculated the averaged value. We show the results in Fig. 5.

9.2 Discussion

As shown in Table 2, the generated prototypes were almost five times larger than the original descriptions. Furthermore, the implementations of the PULL-first and PUSH-first prototypes were significantly different. This suggests that addition of the control-dependent information can bridge a certain part of the gap between the abstraction levels of the architecture models and their implementations. In Fig. 5, the communication load to update a state is higher in PUSH-style than PULL-style, but that to obtain the state is higher in PULL-style than PUSH-style. These results conform to the tendency of the performance expected in Sec. 2. Therefore, we can expect that the generated prototypes can be used to estimate the impact of the selection on the resulting services performance.

10 Related Work

Little work has been done on helping architects to select suitable data transfer methods. In [12], Zhao presented a model of computation for push/pull communications. Compared to our model, the model has a less theoretical basis and does not cover components' behavior. However, some constraint on push and pull combinations (which is discussed in Sec. 5.3) was also discussed in the paper. In the context of parallel graph computation, Besta et. al. [2] exhaustively analyzed the performance of push/pull variants of various graph algorithms.

[3] CPU: E5-1603 v4 2.80GHz, RAM: 32.0GB, JVM: jdk-12, Spring Boot: v2.4.1

Most of the formal architecture models (cf. [1, 6, 8]) are process-centric. The abstraction level of process-centric models is considered lower than ours because process-centric models are control-dependent. In fact, by adding the information about the control-flow to our model, CCS processes can be derived and the equivalence of derived processes can also be proved. In [5], labeled transition systems (LTLs) are used to represent and check behavior of RESTful Web applications. Since the state space of LTL is limited to finite, many temporal properties of the model can be automatically verified, but it does not have sufficient expressive power to represent many kinds of data structure such as natural numbers and lists. In contrast, our model can have infinite state space and can generate executable prototypes.

Dataflow programming [10, 11, 9] has been studied for about five decades. A dataflow program internally uses a directed graph that represents the set of all instructions and dataflow among the instructions. Both the representation and the semantics of dataflow programming are similar to ours, and in fact, our model can be implemented as a dataflow program rather straightforwardly. However, the abstraction level of our model is higher than that of dataflow programming in terms of the following three aspects. First, unlike dataflow programming, in our model, we assume that the states of all resources are synchronously changed and the updating process is not observable. Second, in our model, whether the state of a resource is stored or not is determined by analyzing the description of the model, but in dataflow programming, it should be explicitly specified in the program as a kind of self-loop. Last, each data transfer method in a dataflow program is fixed to PUSH-style.

11 Conclusion

We have presented a formal architecture model to abstract some control-independent parts of architecture designs that are not affected by changes of data transfer methods and an architecture level refactoring for changing data transfer methods. We have also developed an algorithm to generate JAX-RS prototypes from the architecture model and selected data transfer methods, and proved the correctness of the algorithm. Furthermore, we have implemented a graph-based refactoring tool and conducted case studies using the tool. The results indicate that the generated prototypes can be used to estimate the impact of the selection on the resulting services performance.

As future work, we want to develop an implementation level refactoring for changing data transfer methods. Extending our architecture model to capture runtime resource creation and deletion is also an important issue.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. and Method **6**(3), 213–249 (1997)

2. Besta, M., Podstawski, M., Groner, L., Solomonik, E., Hoefler, T.: To push or to pull: on reducing communication and synchronization in graph computations. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. pp. 93–104 (2017)
3. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
4. Joseph Goguen, J.T., Wagner, E.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. Tech. rep., IBM, TJ Watson Research Center (1976)
5. Klein, U., Namjoshi, K.S.: Formalization and automated verification of restful behavior. In: Proceedings of the 23rd international conference on Computer aided verification (CAV'11). pp. 541–556 (2011)
6. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In: Proceedings of Working IEEE/Int'l Federation for Information Processing Conf. Software Architecture. pp. 35–49 (1999)
7. Milner, R.: Communication and concurrency. Prentice-Hall (1990)
8. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: a framework for designing and verifying architectural specifications. IEEE Transactions on Software Engineering **35**(3), 325–346 (2009)
9. Sousa, T.B.: Dataflow programming: concept, languages and applications. In: Proceedings of 7th Doctoral Symposium on Informatics Engineering (2012)
10. Sutherland, W.R.: Online graphical specification of computer procedures. Ph.D. thesis, MIT (1966)
11. Veen, A.H.: Dataflow machine architecture. ACM Comput. Surv. **18**, 365–396 (1986)
12. Zhao, Y.: A model of computation with push and pull processing. Tech. Rep. UCB/ERL M03/51, EECS Department, University of California, Berkeley (2003), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2003/4192.html