Approaching Simple and Powerful Service-Computing

Xingwu Liu
The Institute of Computing Technology,
Chinese Academy of Sciences
xlw@software.ict.ac.cn

Zhiwei Xu
The Institute of Computing Technology,
Chinese Academy of Sciences
zxu@ict.ac.cn

Abstract

Service-computing is the computing paradigm that utilizes services as building blocks for developing applications or solutions. Because simplicity of services, applications, and their interaction are critical for low cost and high productivity of service-computing, this paper tries to explore a proper bound of the simplicity. By proving that any Turing machine is equivalent to the interaction product of three generalized finite automata, we show that the services can be as simple as generalized finite automata, and their interaction can be as simple as interaction product. Since generalized finite automata are equivalent to finite automata which are intuitively very simple, and interaction product is an interaction mechanism imposing few constraints on its components, this result is generally helpful in distributed-system designs for achieving low cost and high productivity.

1. Introduction

Service-computing\textsuperscript{[1-4]}, or service-oriented computing, emerged in the mid-90s and is gaining popularity due to growing interest in Internet-based services. Its goal is to make services network-addressable entities and reusable building blocks that can be orchestrated into usable and efficient service-based applications. Promising to achieve high efficiency and low cost which are among the traditional needs of IT organizations, service-computing has attracted much attention from industry and research community, whose efforts over the last decade has resulted in fruitful progress\textsuperscript{[5-9]}.

In this paradigm, autonomous and platform-independent services are orchestrated and interact to solve problems, so the two elements are individual services and the interaction among them. From the viewpoint of a service provider, the simpler the service is, the easier to develop and maintain it. Hence, ideally, services should be as simple as possible. Likewise, users hope to easily orchestrate service to construct their application, so the applications also should be as simple as possible. Meanwhile, to reduce the cost and complexity of system infrastructure, the interaction among services should be as simple as possible. Then a question arises: how simple can these factors be, yet leaving the whole system powerful enough? Simplicity and powerfulness can be interpreted intuitively as follows. Modeling each service and application as an interactive process described as an automaton, the authors claim that the closer to finite automata a machine in Chomsky hierarchy is, the simpler it is, while the closer to Turing machines, the more powerful. As to interaction, we believe that the less the components are constrained (e.g., the more autonomous and asynchronous they are allowed), the simpler the interaction is.

Our modeling is motivated by many works, especially by \textsuperscript{[10]} and \textsuperscript{[11]}. In \textsuperscript{[10]}, J. Jeng and W. Tsai proposed a finite state machine architectural framework for service-based applications, in which each service is represented by a finite state machine which interacts with one another by event notification facilities. In \textsuperscript{[11]}, services are modeled as processes that are specified by finite state machines, and they communicate through FIFO message queues. The main difference of our model from them lies in two fact. (i) In our model, interaction is done through sharing a finite number of buffers. (ii) Our interaction mechanism imposes few constraints on the components, allowing them to be autonomous and asynchronous.

Closely related to our question is the theory of interaction, which is the focus of an increasing, massive literature\textsuperscript{[12-14]}. As early as in 1939, Turing\textsuperscript{[15]} proposed the model of Turing machines with oracle, called o-
Turing machines, which may be more powerful than Turing machines. J. van Leeuwen et al. [16] considered a much restricted oracle, advice, and proved that advice also brings about computing capability beyond Turing machines. In [17], Goldin et al. defined persistent Turing machines (PTMs) to model interaction, proposed stream language to measure the expressiveness of interaction, and formally proved that PTMs are more powerful than Turing machines. All this indicates that interaction empowers Turing machines with stronger capabilities. Hence, by analog, it may be the case that some simple machines can simulate Turing machines by some form of interaction, justifying our question.

Our main contribution lies in two aspects. Firstly, in service-computing, the services can theoretically be as simple as generalized finite automaton (GFA, defined in section 4), and at the same time, the interaction can bring about computing capability beyond Turing machines. Generally, a partial function. Each term defined by δ, say δ(q, a₀,…, aₙ) = (q', b₀,…, bₙ, d₀,…, dₙ), is called an operation of M, it’s applicable when the current state of M is q and the scanned symbol on tape i is ai for each 0≤i≤n, and its application transits the finite control to state q', rewrites the scanned cell with bi, and causes the ith head to move one step left, one step right, or stand still in the case that di = L, R, or S, for 0≤i≤n.

Without loss of generality, we assume that Q ∩ Γ = ∅.

At any time, the configuration of M is represented by an (n+1)-tuple (a₀,q₀,b₀,…,aₙ,qₙ,bₙ) where for each 0≤i≤n, aᵢ, qᵢ, bᵢ, dᵢ is a finite string starting from the left end of the ith tape and covering all the non-blank cells on that tape with the ith head pointing to the right-most symbol of aᵢ, and qᵢ is the current state of M. Each bᵢ is not unique, in that any number of blanks may be added to or removed from its right. If all the cells to the right of the ith head are blank, bᵢ may be an empty string.

2.2. Turing machine with buffers

Definition 1. A Turing machine with n buffers, called an n-BTM, is an (n+1)-TM (Q, Σ, Γ, δ, q₀, B, F) where δ(q,a₀,…,aₙ) = (q’, b₀,…, bₙ, d₀,…, dₙ) is a partial function from [18] terminology about Turing machines, for example, finite control, tapes and their cells, heads associated with the tapes, scanned symbols, transition functions, and the language-recognizing semantics.

Formally, an (n+1)-TM M is described by a 7-tuple (Q, Σ, Γ, δ, q₀, B, F) where Q is the finite set of states, Σ is the set of input symbols, Γ ⊇ Σ is the finite set of tape symbols, δ: Q × Γⁿ⁺₁ → Q × Γⁿ⁺¹ × {L, R, S} is the transition function, q₀ ∈ Q is the start state, B ∈ Γ · Σ is the blank symbol, and F ⊆ Q is a set of final states.

Generally, δ is a partial function. Each term defined by δ, say δ(q, a₀,…, aₙ) = (q’, b₀,…, bₙ, d₀,…, dₙ), is called an operation of M, it’s applicable when the current state of M is q and the scanned symbol on tape i is ai for each 0≤i≤n, and its application transits the finite control to state q’, rewrites the scanned cell with bi, and causes the ith head to move one step left, one step right, or stay still in the case that di = L, R, or S, for 0≤i≤n.

Without loss of generality, we assume that Q ∩ Γ = ∅.

At any time, the configuration of M is represented by an (n+1)-tuple (a₀,q₀,b₀,…,aₙ,qₙ,bₙ) where for each 0≤i≤n, aᵢ, qᵢ, bᵢ, dᵢ is a finite string starting from the left end of the ith tape and covering all the non-blank cells on that tape with the ith head pointing to the right-most symbol of aᵢ, and qᵢ is the current state of M. Each bᵢ is not unique, in that any number of blanks may be added to or removed from its right. If all the cells to the right of the ith head are blank, bᵢ may be an empty string.

2.2. Turing machine with buffers

Definition 1. A Turing machine with n buffers, called n-BTM, is an (n+1)-TM (Q, Σ, Γ, δ, q₀, B, F) each of whose operations, say δ(q,a₀,…,aₙ) = (q’, b₀,…, bₙ, d₀,…, dₙ), satisfies dᵢ = S for 1≤i≤n. Since for 1≤i≤n, the ith head of M doesn’t move, we simplify tape i into a single cell named buffer i (Figure 1), and reduce δ to a partial function from Q × Γⁿ⁺¹ to Q × Γⁿ⁺¹ × {L, R, S} in an obvious way. Hereafter, the tape of an n-BTM refers to its tape 0, and the head its head 0.

Definition 2. At any time, the configuration of an n-BTM M is an (n+1)-tuple (a₀,q₀,b₀,…,aₙ,qₙ,bₙ) such that (a₀,q₀,b₀,…,bₙ,qₙ) is the configuration of M if it’s regarded as an (n+1)-TM.

![Figure 1. n-BTM](image)

3. Equivalence of a TM to the interaction product of its projections

In this section, let M be an arbitrary (n+1)-TM (Q, Σ, Γ, δ, q₀, B, F), and A be the set of symbols that may appear on its tapes 1 to n, i.e. A is the smallest subset of Γ such that if δ(q, a₀,…, aₙ) = (q’, b₀,…, bₙ, d₀,…, dₙ), then a₁,…, aₙ, b₁,…, bₙ ∈ A.
Let $\Gamma = \{ [a, d] | a \in A, d = \{ L, S, R \} \}$.

### 3.1. Interaction product

Build an $n$-BTM $M_0 = (Q, \Sigma, \Gamma, q_0, \delta_0, B, F)$ in which $\delta_0$ is defined by $\delta_0(q, a_0, \ldots, a_n) = (q', b_0, [b_1, d_1], \ldots, [b_n, d_n], d_0)$, where $q \in Q$, $a \in \Gamma$ for $0 \leq i \leq n$, and $\delta(q, a_0, \ldots, a_n) = (q', b_0, \ldots, b_n, d_0, d_n)$. For notational convenience, let $Q_0 = Q$, $\Sigma_0 = \Sigma$, and $\Gamma_0 = \Gamma \cup \Gamma'$.

And build $n$ 1-BTM $M_i = \ldots = M_i = (\{ r, s \}, \alpha - \{ B \}, A \cup \Gamma, \delta_i, r, B, r)$ where

1. $r$ is an arbitrary symbol not in $A \cup \Gamma$;
2. $\delta_i$ is defined by $\delta_i(r, a, [b, d]) = ([F', b, [b, d], d])$ for $a, b \in A$ and $d \in \{ L, S, R \}$, and $\delta_i([F, a, b]) = ([r', a, a, S])$ for $a \in A$ and $b \in \Gamma'$.

Again for notational convenience, let $Q_i = \{ r, F \}$, $\Sigma_i = \alpha - \{ B \}$, $\Gamma_i = A \cup \Gamma$, and $\delta_i = \delta_i$ for $1 \leq i \leq n$.

**Definition 3.** For each $0 \leq i \leq n$, $M_i$ is referred to as the $i$th projection of $M$.

In terms of the projections of $M$, we construct a system $M_e$ (Figure 2) by sharing the $i$th buffer of $M_0$ with the buffer of $M_i$ for $1 \leq i \leq n$. Given an input string $a \in \Sigma^*$, $M_e$ works as follows. At first, $M_0$ is initialized with input $a$, and the other projections are initialized with empty input. Then, each projection runs according to its transition function, and interacts with others by reading/writing the shared buffers. The system satisfies

1. Asynchrony: how fast a projection proceeds is independent of the others;
2. Autonomy: at any time what a projection can do depends on nothing but its own state, scanned symbol, and buffered symbol(s), and if it has no applicable operation it does nothing;
3. Non-isolation: in each operation, $M_0$ rewrites $n$ buffers not necessarily simultaneously, and once a buffer-rewriting is finished the effect is immediately visible to other projections, even though the operation as a whole has not ended.

![Figure 2. Interaction product of $M_0$, $M_1$, ..., $M_n$](image)

**Definition 4.** The system $M_e$ is called the interaction product of $M_0$, $M_1$, ..., and $M_n$, and is denoted by $\times(M_0, \ldots, M_n)$.

Roughly speaking, $M_e$ simulates $M$ with each projection simulating $M$'s behavior on that tap and with the finite control of $M_0$ simulating that of $M$. This will be clarified in the following.

### 3.2. Configuration of $M_e$

In order to formally describe how $M_e$ works, we develop a notation for its configurations.

**Definition 5.** The configuration of $M_e$ at time $t$ is represented by a $(2n+1)$-tuple $C = (a_0, b_0, \ldots, a_n, b_n, 0, 0, \ldots, 0)$ where for $0 \leq i \leq n$ and $0 \leq j \leq n$, $a_i, b_i \in \Gamma^*$, $p_i \in Q$ is a state of $M_i$, and $b_i$ is a 5-tuple $(u_j, v_j, x_j, y_j, z_j)$ in which $u_j, v_j, w_j \in \Gamma_i$ and $x_j, z_j \in \{ 0, 1 \}$.

Intuitively, the configuration $C$ at time $t$ has the following meanings. For $0 \leq i \leq n$, $a_i b_i$ is the first entry of the configuration of $M_i$ if $M_i$ has no operation in progress, and otherwise is that of the coming configuration of $M_i$. For $1 \leq j \leq n$, $x_j = 1$ means that $M_j$ has an operation in progress to rewrite buffer $i$ with $w_j$, and otherwise $x_j = 0$; $z_j = 1$ means that $M_i$ has an operation in progress to rewrite buffer $i$ with $y_j$ and otherwise $z_j = 0$; $u_i$ is the symbol in buffer $i$ at that time.

**Definition 6.** Given an input string $a \in \Sigma^*$ of $M_e$, assume that $ab = c\beta$ where $c$ is a symbol in $\Sigma \cup B$.

Then by the initialization of $M_e$, it’s reasonable to define the starting configuration $I_e$ by $I_e = (c, \beta, \ldots, 0)$.

**Definition 7.** Let $C = (a_0, b_0, \ldots, a_n, b_n, 0, 0, \ldots, 0)$. Its busy set is $BS(C) = \{ i | x_i = 1 \}$. $M_0$ is busy in $C$ if $BS(C) \neq \emptyset$, $M_0$, for $i \neq 0$, is busy if $z_i = 1$, and they are said to be free otherwise. The 0-associate of $C$ is an $n+1$-tuple $(a_0, b_0, a_1, \ldots, a_n)$ where $a_0 = w_0$ if $x_0 = 1$, and $a_1 = w_1$ otherwise, for $1 \leq j \leq n$, the i-associate, $1 \leq i \leq n$, is a couple $(a_0 b_i, a_i)$, where $a_i = v_i$ if $z_i = 1$, and $a_0 = w_i$ otherwise. An operation of $M_i$, $0 \leq i \leq n$, is applicable to $C$ if $M_i$ is free and the operation is applicable to the configuration of $M_i$ which is identical with the i-associate of $C$.

### 3.3. Dynamics of configurations of $M_e$

**Definition 8.** An action vector $v = (o_0, a_1, \ldots, a_n)$ of $M_e$ is an $n+1$-tuple where

1. $o_0$ is either $c$, an operation of $M_0$, or a non-empty subset of $\{ 1, 2, \ldots, n \}$,
2. for $1 \leq i \leq n$, $o_i$ is either $c$, 1, or an operation of $M_i$.

An action vector models the behavior of all projections at time $t$. Intuitively, at that time, if $M_0$ rewrites some buffers, then $o_0$ is the set of the indices of those buffers; if $M_i$ where $i \neq 0$, rewrites its buffer, then $o_i = \ldots$
1; if \( M_i, 0 \leq i \leq n \), starts an operation \( op \) then \( o_i = op \); otherwise, the entries are \( c \).

**Definition 9.** An action vector \( v \) is applicable to a configuration \( C \) of \( M_i \) if for every \( 0 \leq i \leq n \) one of the conditions is satisfied: (1) \( o_i = c \); (2) \( o_i = 1 \) and \( M_i \) is busy in \( C \); (3) \( i = 0 \) and \( o_i \) is a nonempty set of \( BS(C) \); (4) \( o_i \) is an operation of \( M_i \) applicable to \( C \).

**Definition 10.** An action vector \( v \) is conflict-free if

1. When \( o_0 \) is an operation, \( o_i = c \) holds for all \( 1 \leq i \leq n \), and
2. When \( o_0 \) is a non-empty subset of \( \{1, 2, \ldots, n\} \), \( o_i = c \) holds for all \( i \in o_0 \).

**Definition 11.** Assume \( v = (o_0, o_1, \ldots, o_n) \) to be a conflict-free action vector such that it is applicable to configuration \( C = (c_0, c_1, \ldots, c_n) \) where \( a_i = (u_i, w_i, x_i, y_i, z_i) \) for \( 1 \leq i \leq n \). Applying \( v \) to \( C \) results in the configuration \( C' = (c_0', c_1', \ldots, c_n') \) which is defined by the following rules and denoted by \( C' = \Gamma(v, C) \):

1. If \( o_0 \) is an operation which transits \( M_0 \) to configuration \( (a_0', p_0', r_0', w_0', x_0', y_0', z_0') \), then \( c_0' = (p_0', r_0', b_0', \phi_0', \lambda_0') \), and \( \alpha_0' = (u_0, w_0', 1, y_0', z_0' \mid 1 \leq i \leq n) \);
2. If \( o_0 \) is a non-empty subset of \( BS(C) \), then for every \( i \in o_0 \), \( \alpha_i' = (w_i, B, 0, y_i, z_i) \);
3. For \( 1 \leq i \leq n \), if \( o_0 \) is an operation of \( M_i \) which transits \( M_i \) to configuration \( (a_i', p_i', r_i', w_i', x_i', y_i', z_i) \), then \( c_i' = (p_i', r_i', b_i', \phi_i', \lambda_i') \), and \( \alpha_i' = (u_i, w_i, x_i, y_i', 1) \);
4. For \( 1 \leq i \leq n \), if \( o_i = 1 \), then \( \alpha_i' = (y_i', w_i, x_i, B, 0) \);
5. \( C \) and \( C' \) share the other entries.

**Remark 1.** Since \( v \) is conflict-free, \( C \) is well defined. And we can derive a mapping \( f \) such that, for all \( 1 \leq i \leq n \), if \( o_i \neq c \) then \( (c_i', \alpha_i') = f(c_i, \alpha_i, o_i) \), and a mapping \( g \) such that if \( o_0 \) is a non-empty subset and \( i \in o_0 \), \( \alpha_i = g(o_i) \).

### 3.4. Basic properties of valid configurations

**Definition 12.** A configuration \( C \) of \( M_i \) is said to be reconciled if it satisfies the conditions

1. If \( M_i \) is busy for some \( i \neq 0 \), then \( M_0 \) has no applicable operation even if it is free, and
2. If \( M_0 \) is busy and \( i \in BS(C) \), then \( M_i \) is free and has no applicable operation.

**Lemma 1.** Any action vector applicable to a reconciled configuration of \( M_i \) is conflict-free.

Proof: Assume that \( C \) is a reconciled configuration of \( M_i \), and that \( v = (o_0, o_1, \ldots, o_n) \) is an action vector applicable to \( C \).

If \( o_0 \) is an operation, by the assumption that \( C \) is reconciled, \( M_i \) is free for \( 1 \leq i \leq n \). Let \( c_i \) be the symbol in buffer \( i \). Since either \( c_i \in A \) or \( c_i \in \Gamma_i \), \( o_0 \) and \( o_i \) can’t be an operation at the same time, so \( o_i = c \) for \( 1 \leq i \leq n \).

If \( o_0 \) is a nonempty subset of \( \{1, 2, \ldots, n\} \) and \( i \in o_0 \), then \( i \in BS(C) \), and \( M_i \) is free and has no applicable operation, which implies that \( o_i = c \).

**Definition 13.** A run \( R \) is a finite or infinite sequence of conflict-free action vectors. \( R \) is applicable to a configuration \( C \) if the vectors in it are applicable in turn starting from \( C \). If \( R \) is finite, the resulting configuration is denoted by \( <C, R> \). A configuration \( R \) is valid if there is a string \( a \in \Sigma^* \) and a finite run \( R \) such that \( C = <I, R> \).

**Remark 2.** For the sake of generality, no restriction is imposed on runs except that not all the entries of a vector in a run can be \( c \).

**Lemma 2.** Any valid configuration is reconciled.

Proof: Consider \( C = \{c\} \), \( R \) where \( I \) is a starting configuration and \( R \) is a finite run.

1. If \( M_i \), where \( 1 \leq i \leq n \), is busy and \( M_0 \) is free in \( C \), then we show that \( M_0 \) has no applicable operation.

Let \( R = v_1 v_2 \ldots v_n \), \( v_i = (o_i, a_{i1}, \ldots, a_{ij}) \), \( C_0 = I \), and \( C = (v_1, v_2, \ldots, v_n) \). Let \( C = (o_0, a_{01}, \ldots, a_{0j}) \) for \( 0 \leq j \leq m \). Assume that \( M_i \) is busy and \( M_0 \) is free in \( C_0 \). Let \( k \leq m \) be the maximal number such that \( a_{ij} \) is an operation of \( M_j \), then it's obvious that \( a_{ij} \in \Gamma_j \). Since \( M_i \) is busy in \( C_{ij} \), \( o_{ij} = c \) for \( k < h \leq m \). Let \( j \) be the maximal number such that \( a_{ij} \) is an operation of \( M_j \). Since \( M_j \) is free in \( C_{ij} \), there is a number \( l \), \( j < h \leq m \), satisfying that \( o_{ij} \) is a set of numbers and that \( i \in o_0 \).

Case 1: \( l > k \). Since \( o_{ij} \in \Gamma_j \), it's easy to see that \( u_{ij} \in \Gamma_j \). Because \( o_{ij} = c \) and \( i \in BS(C_{ij}) \) for \( k < h \leq m \), we have \( u_{ij} \ldots u_{i+1j} = u_{ij} \in \Gamma_j \), so \( M_0 \) has no operation applicable to \( C_{ij} \).

Case 2: \( l = k \). Then \( o_{ij} \) is a set of numbers, \( i \in o_0 \), and \( a_{ij} \) is an operation of \( M_j \), which is contradictory to the premise that \( v_i \) is conflict-free.

Case 3: \( l = k \). Since \( o_{ij} = c \) and \( i \in BS(C_{ij}) \) for \( k < h \leq m \), we have \( u_{ij} \ldots u_{i+1j} = u_{ij} \in \Gamma_j \), so \( M_0 \) has no operation applicable to \( C_{ij} \).

The detail is omitted here.

**Remark 3.** By Lemma 1 and 2, any action vector that is applicable to a valid configuration is conflict-free. Though the applicability is locally determined and the freedom of conflicts is a global property, local "legality" guarantees global "consistence" in the context of valid configurations. Hence, we redefine a run of \( M_i \) as a sequence of action vectors since only valid configurations are considered in the rest of the paper.

**Remark 4.** Lemma 2 also shows that \( M_i \) works without buffer-writing conflict and thus with definite
3.5. Trace uniqueness of maximal runs

Definition 14. A run \( R \) is maximal with regard to a valid configuration \( C \) if it is applicable to \( C \) and is not a proper prefix of any such run.

Of course, if \( R \) is infinite, it’s maximal with regard to \( C \) if and only if it’s applicable to \( C \).

Definition 15. Given a run \( R = v_1v_2...v_m \) where \( v_i = (o_{i0}, ..., o_{in}) \), its trace, denoted by \( T(R) \), is the subsequence of \( o_{i0} o_{i1} o_{i2}...o_{in} \) only including all the entries that are operations of \( M_0 \).

It’s obvious that \( T(\bar{R} \cup \bar{R'}) = T(\bar{R}) \cup T(\bar{R'}) \), where the operator \( \bar{\cdot} \) means the concatenation of two sequences.

Definition 16. Given a run \( R = v_1v_2...v_m \) where \( v_i = (o_{i0}, ..., o_{in}) \) for \( i \geq 1 \), we have the following 4 Lemmas.

Lemmas 3. Assume that there is a number \( 1 \leq j \leq n \) and \( j \geq 0 \) such that \( o_0 \) is a subset of \( \{1, 2, ..., n\} \) and \( o_j \neq \emptyset \). Let \( R' = v_1v_2...v_j \) be a run \( o_0 c_1...o_j \), where \( R'' \) is the run \( o_0 c_1...o_j o_0 \). If \( R'' \) is applicable to a valid configuration \( C \), then \( o_j \) is applicable to \( C \).

Proof: Let \( C_i = (o_{i0}, ..., o_{in}) = (o_{i0}, ..., o_{in}) \) for \( k \geq 0 \). Recall the mappings \( f \) and \( g \) defined in Remark 1. Because \( o_0 \) is a subset of \( \{1, 2, ..., n\} \), it follows that \( a_{i,j} = g(a_{i,j}) \) for all \( h \in o_0 \). For \( 1 \leq h \leq n \) such that \( o_{j} \neq \emptyset \), we have \( (c_{j0, a_{j}}) = f(c_{j0, a_{j}}, o_{j}, o_{j}) \). The other entries of \( C_i \) are the same as those of \( C_{j+1} \).

On the other hand, it follows from \( o_0 c_1...o_j \subset BS(C_{j+1}) \) that \( o_0, c_1, ..., c_j \) is applicable to \( C_{j+1} \). Let the resulting configuration be \( C' = (c_{j0}, c_{j1}, c_{j2}, ..., a_{j}) \). Then \( a_{j} = g(a_{j}) \) holds for every \( k \in o_{j0} \), and the other entries of \( C' \) are the same as those of \( C_{j+1} \).

Lemmas 4. Assume that there is a number \( j \) such that \( o_0 = c_0, o_{j+1} = o_{j+2} = ... = o_{j+n} = c_0 \) for all \( i > 0 \). If \( R \) is applicable to a valid configuration \( C \), so is \( R' = v_1v_2...v_j \) if \( R' = v_1v_2...v_j \) is also applicable to \( C \).

The proof is similar to that of Lemma 3 and is omitted here.

Lemma 5. Assume there is a number \( 1 \leq j \leq n \) and \( j \geq 1 \) such that \( o_{j0} = o_{j1} = o_{j2} = ... = o_{jn} \). Let \( R'' = v_1v_2...v_j = (o_{j0}, o_{j1}, o_{j2}, ..., o_{jn}) \). If there is a number \( h \neq j \) such that \( o_{j0} \neq o_{j1} \), \( o_{j2} = ... = o_{jn} \) if \( R'' = v_1v_2...v_j \) otherwise. If \( R \) is applicable to a valid configuration \( C \), then so is \( R' = v_1v_2...v_j \).

The proof is similar to that of Lemma 3 and is omitted here.

Lemma 6. Assume there is a number \( j \geq 1 \) such that \( o_{j0} = o_{j1} = o_{j2} = ... = o_{jn} \) for \( i > 0 \), which is applicable to a valid configuration \( C \), and \( o_{j} \neq o_{j1} \), \( o_{j} = o_{j1} \) for all \( 1 \leq j \leq n \). If \( R \) is applicable to a valid configuration \( C \), then \( o_{j} \neq o_{j1} \), \( o_{j} \neq o_{j2} \), etc.

The proof is similar to that of Lemma 3 and is omitted here.

Remark 5. In Lemma 3-6, \( T(R) = T(\bar{R}) \).

Lemma 7. Given a run \( R = v_1v_2...v_m \) where \( v_i \neq (o_{i0}, ..., o_{i-n}) \) for \( i > 0 \) and \( v_i \neq (o_{i0}, ..., o_{i-n}) \). If \( R' \) is applicable to a valid configuration \( C \), then \( R' = v_1v_2...v_m \).

Remark 6. A run \( R = v_1v_2...v_m \) is said to be \( C \) if it’s applicable to a valid configuration \( C \), and \( R \) is \( C \) if it’s applicable to a valid configuration \( C \).

Corollary 1. Any two runs that are maximal with regard to a valid configuration \( C \) have the same trace.

This corollary follows immediately from Lemma 3-6 and Remark 5.

3.6. The expressiveness of \( M_x \)

Definition 17. Given a string \( a \in \Sigma^* \), a run of \( M_x \) accepts \( a \) if it has a prefix \( \bar{R} \) such that the state of \( M_0 \) in \( \bar{R} \) is final. A string is said to be accepted by \( M_x \) if it’s accepted by some run.

Remark 7. Let \( R_1 \) and \( R_2 \) be two runs of \( M_x \) such that \( T(R_1) = T(R_2) \), then \( R_1 \) accepts a string if and only if \( R_2 \) does.

Definition 18. Given configurations \( C_i = (o_{ib}, o_{ib}, o_{ib}, o_{ib}) \) of \( M \) and \( C_i = (o_{ib}, o_{ib}, o_{ib}, o_{ib}) \) of \( M \), where \( q \in Q \) and \( p \in Q \), they are to be said to be consistent (denoted by \( C_1 \sim C_2 \) if \( 0 \leq t < n \) and \( 1 \leq n < \), the following equations hold: \( a_i = y_i, b_i = z_i \) if some blanks are added to or removed from the right end of \( a_i, q = p_0, p_0 = p \), and if \( y_i = z_i = 0 \).

Lemma 8. Given a configuration \( C_1 \) of \( M \) and a configuration \( C_2 \) of \( M_x \) such that \( C_1 \sim C_2 \), then the following statements hold.
(1) If there is an operation $op$ of $M$ which is applicable to $C_1$, then there is a run of $M_{op}$ such that $<C_1, op>=<C_2, R>$; and

(2) If there is a run of $M_{op}$ which is applicable to $C_2$, then there is a run of $M_{op}$ also applicable to $C_1$ and an operation $op$ of $M_{op}$ applicable to $C_2$, such that $<C_1, op>=<C_2, R>$.

Proof: Let $C_1=(q_0, q_1, q_2, \ldots, q_n)$, and $C_2=(q'_0, q'_1, q'_2, \ldots, q'_n)$ for each $1 \leq i \leq n$. Now construct a run $R=v_1v_2\cdots v_n$, where $v_i=(q_i, q_j)$ for $1 \leq i \leq 6$. The entry $q_i=q_j$ is the operation $\delta(q, q_0, u_1, \ldots, u_{n-1}, q_{n-1}) = (q', q_0)$, where $q'$ is the symbol left to $q$, and is the rightmost symbol of $q$. Construct a run $R$ just the same as that in the proof of statement 1. By the definition of $M_{op}$, $\delta(q, q_0, u_1, \ldots, u_{n-1}, q_{n-1}) = (q', q_0)$ is an operation of $M$.

Theorem 1. $M$ and $M_{op}$ accept the same language.

Proof: The theorem follows from the following facts. (1) By Remark 7, a maximal normal run exists and is unique. (2) By Corollary 1, if $\delta(q, q_0, a, \ldots, a_n)$ is accepted by $M$, and only if it’s accepted by a maximal normal run with regard to $L_0$. (3) Repeatedly apply Lemma 7 to show that $R$ is applicable to $C_2$, that $op$ is applicable to $C_1$, and that $<C_1, op>=<C_2, R>$.

4. Main result

4.1. Generalized finite automata and their interaction product

Definition 19. A 1-TM $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is an automaton (FA) if each operation $\delta(q, a)=\gamma$, $b$, $d$ satisfies $a=b$, and if for any $q\in Q$, $\delta$ is not defined

for $(q, B)$. For notational simplicity, $M$ is denoted by $(Q, \Sigma, \delta, q_0, F)$, where $\delta: Q\times\Sigma\rightarrow Q\times\{L, R, S\}$ defines $\delta(q, a)=(q', d)$ for each operation $\delta(q, a)=(q', a, d)$. This FA is consistent with the two-way finite automaton in [18]. A string $q\in \Sigma^*$ is accepted by $M$ if with $\alpha$ as input, $M$ moves its head beyond the right end of the input string and halts at a final state.

Definition 20. An $n$-BTM $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, $\delta\in\Sigma^*$, is a generalized finite automaton (GFA) if each operation, say $\delta(q, a_0, \ldots, a_n)$, satisfies $a_0=b_0$. For notational simplicity, each operation $\delta(q, a_0, \ldots, a_n)$ is simplified as $\delta(q, a_0, \ldots, a_n)$. Of course, the GFA is said to be singular. A GFA accepts a string $\alpha\in \Sigma^*$ if with $\alpha$ as input it reaches a final state.

Theorem 2. Given a Turing machine, there are two GFAs $M_0$ and $M_1$, such that $M\equiv(M_0, M_1, M_2)$.

Proof: By Theorem 7.9 in [18], any Turing machine is equivalent to a two-counter machine. Since a two-counter machine has three projections each of which is a GFA, the theorem immediately follows from Theorem 1.

4.2. The expressiveness of GFAs

The following of this section is devoted to the equivalence of a GFA to an FA, which implies that extending FAs with an infinite tape and a finite number of readable and writable buffers cannot extend their computational capability.

Lemma 8. GFAs are equivalent to singular GFAs.

The proof is almost obvious and is omitted here.

Consider a singular GFA $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where $\delta$ is of the form $\delta: Q\times\Gamma\rightarrow Q\times\{L, R, S\}$.

Definition 21. A $flow$ is a sequence $a_1a_2\cdots a_n$ whose length is greater than 1 and where $a_i\in Q$ if $i=1 \mod 2$, $a_i\in \{L, R, S\}$ if $i=j \mod 2$, and $\delta(a_{k+1}, B) = (a_{k+2}, a_{k+1})$ for $k\geq 1$. The sequence $a_1a_2\cdots a_n$, where $a_1$ is the operation $\delta(a_{k+1}, B) = (a_{k+2}, a_{k+1})$ for $k\geq 1$, is called the associated run of the flow.

Definition 22. An expanding flow is a flow $a_1a_2\cdots a_n$ such that for any $n$, the number of $R$’s in sequence $a_1a_2\cdots a_n$ is less than that of $L$’s. If for some $n$ the numbers are equal, the flow $a_1a_2\cdots a_n$ is called an expanding cycle.

Definition 23. For $q, q'\in Q$, $q$ is reducible to $q'$ if there is an expanding cycle starting with $q$ and ending with $q'$.

Now construct an FA $M'=(Q', \Sigma', \delta', q_0', F')$, where $Q'=Q\cup \{\overline{q}\}$, $\Sigma'=\Sigma\{\overline{q}\}$, $q_0'=q_0$, $F'=\{\overline{q}\}$, and $\delta': Q'\times\Sigma'\rightarrow Q'\times\{L, R, S\}$ is defined as follows. For each $(q,
Given a singular GFA is equivalent to an FA. Given a singular GFA $M$, construct $M'$ as above. Since $M'$ is an FA, $L_M$ is a regular language (Theorem 2.4 in [18]). Since $L$ is also regular and regularity is closed under intersection (Theorem 3.4 in [18]), $L_M \cap L$ is regular too.

Now consider a mapping $\varphi: \Sigma \cup \{\}$ maps $a$ to $a$ for $a \in \Sigma$, and maps $\$$ to the empty string. By Lemma 9, this mapping determines a homomorphism from $L_M \cap L$ onto $L_M$. Because regularity is closed under homomorphism and $L_M \cap L$ is regular, $L_M$ is a regular language. By Theorem 2.3 in [18], there is an FA that accepts $L_M$.

### 4.3. An answer to our question

Now, By Lemma 10, a GFA is intuitively quite simple, so it ready to answer the question stated in Introduction.

**Theorem 3.** As far as computational power is concerned, in service-computing, each service can be as simple as a GFA, and the interaction mechanism can be as simple as interaction product, leaving the whole systems as powerful as Turing machines.

### 5. Conclusion

In service-computing, problems are typically solved by some interacting multiple services. For the sake of low cost and high productivity, the individual services, their interaction, and their orchestration are supposed to be as simple as possible, meanwhile keeping the whole system powerful enough. To explore how simple the factors can be, this paper focuses on the interaction mechanism between services, proving that any Turing machine can be simulated by the interaction product of three GFAs. Since GFAs are as simple as finite automata, and interaction product is an interaction mechanism imposing few constraints on its components, a proper answer to this problem is obtained.

Generally, our result identifies GFAs and interaction product as the elements of computation. Thus, as to computational capability, any distributed computing platform such as Grid needs only support GFAs and their interaction product. The asynchrony, autonomy, and non-isolation of interaction products, together with the fact that GFAs are as simple as FAs, help achieve low cost and high productivity of these platforms.

### References


Appendix

Lemma 9. \( L_{X} \cap L = \{ \alpha \Sigma | \alpha \in L_{M} \} \) where \( L = \{ \alpha \Sigma | \alpha \in L_{X} \} \) and \( L_{X} \) denotes the language accepted by a machine \( X \).

Proof: For an input string \( \alpha \beta \in \Sigma^{*} \) where \( \beta \in \Sigma^{*} \), we show that \( M \) accepts \( \alpha \) if and only if \( M \) accepts \( \beta \).

If a run \( o_{1}, o_{2}, \ldots, o_{n} \) of \( M \) accepts \( \alpha \), we have a run \( o_{1} o_{i_{1}} \ldots o_{i_{n}} \) of \( M \) for \( i_{1} = 1 \) and for each \( 1 \leq k \leq n \), the following conditions are satisfied:

1. If \( o_{i} \) is \( \delta(q, a) = (q', d) \) where \( q \in F \cup \{ q \} \) then \( i_{k+1} = i_{k} \).
2. If \( o_{i} \) is \( \delta(q, a) = (q', d) \) where \( a \neq S \) and \( q' \notin S \) then \( o_{i} \) is \( \delta(q, a) = (q', d) \) and \( i_{k+1} = i_{k} + 1 \).
3. If \( o_{i} \) is \( \delta(q, S) = (q', L) \) then \( o_{i} \) is \( \delta(q, B) = (q', L) \).
4. If \( o_{i} \) is \( \delta(q, S) = (q, R) \) then \( o_{i} \) is \( \delta(q, S) = (q, R) \) and \( q \notin F \) and \( q \) is the associated run of the expanding cycle starting with \( q \).
5. If \( o_{i} \) is \( \delta(q, S) = (q, R) \) where \( q \notin F \), then \( q \) must be reducible to state \( q' \), and \( o_{i} o_{i_{1}} \ldots o_{i_{n}} \) is the associated run of the expanding cycle starting with \( q \) and ending with \( q' \).

If \( i_{n+1} = i_{1} \), then obviously \( q_{n} \in F \), so \( q_{0} \in F \) and \( M \) accepts \( \beta \).

On the other hand, if \( M \) accepts \( \beta \) by a run \( o_{a_{2}} \ldots o_{a_{n}} \) where \( n \geq 1 \), assume that there is no proper prefix of the run that also accepts \( \beta \). Construct a run of \( M \) through the following steps.

1. \( h = k = 0 \);
2. If \( o_{0} \) is \( \delta(q, a) = (q', d) \) where \( a \neq B \), then \( o_{0} \) is \( \delta(q, a) = (q', d) \), \( h = h + 1 \), and \( k = k + 1 \);
3. If \( o_{0} \) is \( \delta(q, B) = (q', L) \), then \( o_{0} \) is \( \delta(q, S) = (q', L) \), \( h = h + 1 \), and \( k = k + 1 \);
4. If \( o_{0} \) is \( \delta(q, B) = (q', S) \) or is \( \delta(q, B) = (q', R) \), then \( h_{0} \) be \( \max \{ |h| \leq h \leq n \} \) such that \( o_{i} \) is \( \delta(q, B) = (q_{i}, b_{i}) \) for \( h = h_{i} \) and \( b_{i} = b_{i+1} \ldots b_{i+1} \) is an expanding flow, \( o_{i} \) be \( \delta(q_{i}, S) = (q_{i}, R) \) if \( q_{i} \in F \), and be \( \delta(q_{i}, S) = (q_{i+1}, S) \) if \( q_{i+1} \notin F \), \( h = h_{0} + 1 \), and \( k = k + 1 \);
5. If \( h = n \), go to step 2;
6. If \( \alpha_{i} \) is \( \delta(q, a) = (q', d) \) where \( q' \notin F \), then \( o_{i} \) is \( \delta(q', c) = (q', S) \) for an appropriate \( c \), and \( k = k + 1 \).

Remark: In step 4, if \( q_{i} \notin F \), then \( q_{i} \) must be an expanding cycle, which guarantees that \( o_{i} \) is well defined there.

It is straightforward to prove that \( o_{1} o_{2} \ldots o_{n} \) is applicable to \( M \) with \( \alpha \) as input, and that if \( o_{i} \) is \( \delta(q, c) = (q', d) \), \( q' = q_{i} \).

Consequently, \( M \) can execute \( \delta(q, c) = (q', R) \), where \( c \) ranges over \( \Sigma^{*} \), for a certain times until its head moves beyond the right end of \( \alpha \).

Thus, \( M \) accepts \( \alpha \).