# Automata and Processes on Multisets of Communicating Objects

Linmin Yang[1], Yong Wang[2], and Zhe Dang[1]

[1]School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA
[2]Google Inc., Mountain View, CA 94043, USA

**Abstract.** Inspired by P systems initiated by Gheorghe Pãun, we study a computation model over a multiset of communicating objects. The objects in our model are instances of finite automata. They interact with each other by firing external transitions between two objects. Our model, called a service automaton, is intended to specify, at a high level, a service provided on top of network devices abstracted as communicating objects. We formalize the concept of processes, running over a multiset of objects, of a service automaton and study the computing power of both single-process and multiprocess service automata. In particular, in the multiprocess case, regular maximal parallelism is defined for inter-process synchronization. It turns out that single-process service automata are equivalent to vector addition systems and hence can define nonregular processes. Among other results, we also show that Presburger reachability problem for single-process service automata is decidable, while it becomes undecidable in the multiprocess case. Hence, multiprocess service automata are strictly more powerful than single-process service automata.

**Key words:** P system, service automata, Presburger reachability, network processes.

## 1 Introduction

Network services nowadays can be viewed as programs running on top of a (possibly large) number of devices, such as cellular phones, laptops, PDAs and sensors. How to design and implement such programs has become a central research topic in areas like pervasive computing [15, 19], a proposal of building distributed software systems from (a massive number of) devices that are pervasively hidden in the environment. In fact, such a view has already embedded in algorithmic studies inspired from ant colonies (where each ant resembles a communicating device in our context) [4, 7, 8], as well as in more recent studies on P systems, a biologically inspired abstract computing model running on, in a simplest setting, multisets of symbol or string objects [13, 14].

As an unconventional computing model motivated from natural phenomena of cell evolutions and chemical reactions, P systems were initiated by Gh. Pãun [13, 14] nine years ago. A P system abstracts from the way living cells process chemical compounds in their compartmental structures. Thus, regions defined by a membrane structure contain objects that evolve according to given rules. The objects can be described by symbols or by strings of symbols, in such a way that multisets of objects are placed in

regions of the membrane structure. The membranes themselves are organized as a Venn diagram or a tree structure where one membrane may contain other membranes. By using the rules in a nondeterministic and maximally parallel manner, transitions between the system configurations can be obtained. A sequence of transitions shows how the system is evolving. Objects in P systems are typed but addressless (i.e., the objects do not have individual identifiers), which is an attractive property for modeling high-level networks.

Inspired by P systems, we introduce an automata-theoretic model for the programs over network devices, called service automata, to specify services running over communicating objects (which are an abstraction of, e.g., network devices mentioned earlier). Our model is at the high-level. That is, the communicating objects are typed but addressless. In other words, unique identifiers such as IP addresses for network devices are left (and of course also necessary) for the implementation level. For instance, in a fire truck scheduling system, which is also an example used throughout our paper, a fire emergency calls for one or more trucks that are currently available. In this scenario, exactly which truck is dispatched is not so important as long as the truck is available. Hence, a service automaton runs on multisets of communicating objects. This also resembles traditional high-level programming languages that run on a memory in the sense that a variable is often mapped with a concrete memory address only at compile time.

In a service automaton, *(communicating) objects* are logical representations of physical devices and entities in a network. Functions of such a device or entity are abstracted as an automaton specified in the correspondent object. In this paper, we mostly study the case when the automaton is of finite-states, i.e., a finite automaton (FA). As we mentioned earlier, objects are typed but addressless in our model and the type of an object is the FA associated with it. In other words, a service automaton runs on a multiset of objects, which are modeled as finite automata.

We depict a service automaton as a finite diagram consisting of a number of big circles. Each circle represents an object type that is an FA whose state transitions, called *internal transitions*, are drawn inside the circle. Notice that an unbounded number of objects could share with the same object type. Communications between objects are specified by *external transitions*, each of which connects two (internal) transitions. An example service automaton is depicted in Fig. 1.

We could impose an initial constraint on a service automaton to explicitly set the number of objects of certain types at the initial time, while the number of objects of other types are not specified. For a service automaton without an initial constraint, the number of objects for each type is not specified. That is, the automaton can run on any multiset of objects that are of the object types specified in the diagram of the service automaton. The service automaton starts from an initial object (of a predefined initial object type) and, at this moment, we say that the object is *active*. Roughly speaking, at each step, the service automaton runs as follows. Suppose that the current active object $O$ is of type $A$ and is at state $q$. At the step, either an active object fires a purely internal transition (that is an internal transition not connected by any external transitions in the diagram of the service automaton) from its current state $q$ to a new state and remains being active, or the active object $O$ communicates with another nondeterministically chosen object $O'$ (we use $B$ to denote its type and $p$ to denote its current state) through

firing an external transition. Suppose that the external transition is $r$. To ensure $r$ firable, we further require that, on the diagram of the service automaton, the $r$ connects from an internal transition $t_A$ inside the big circle of type $A$ to an internal transition $t_B$ inside the big circle of type $B$. Furthermore, the two internal transitions start with the current states of the two objects $O$ and $O'$, respectively. On firing the external transition, both objects $O$ and $O'$ fire the two internal transitions, respectively and simultaneously. After firing the external transition, the current active object becomes $O'$ and the object $O$ is no longer active.

Actually, we can view an active object as one holding a token. When an external transition (between two objects) is fired, it can pass the token to the other object. When we follow the flow of the token, we can define a *process* of the service automaton as a sequence (of labels) of transitions that the token is passing through. Hence, the service defined by the service automaton is the set (i.e., language) of all such processes. In the paper, we show that service automata and vector addition systems are equivalent and hence can define nonregular services. We also discuss other variations and verification problems of service automata. One interesting open question is that we currently do not know whether there is a nontrivial subclass of service automata that only define regular services.

In the service automaton given above, there is only one object being active at any time (i.e., there is only one token), and hence it is a single-process service automaton. In the paper, we also study *multiprocess service automata*, where there are multiple active objects at any time; i.e., there are multiple tokens, each of which corresponds to a process. Inter-process communication is also defined through our notion of regular maximal parallelism among processes, which generalizes Păun's [14] classic maximal parallelism as well as other derivation modes [6, 10] in the context of P systems. One of our results shows that multiprocess service automata are strictly stronger than (single-process) service automata. We also study variations and verification problems for multiprocess service automata.

Our service automata, in their current form (where each object type specifies an FA), can be treated as a variation of P systems where each object is a pair of a symbol and a state. Roughly speaking, an external transition that connects from the internal transition $q \rightarrow q'$ in an automaton of type $A$ to the internal transition $p \rightarrow p'$ in an automaton of type $B$ can be depicted in a P system rule in the following form:

$$\bar{A}_q B_p \rightarrow A_{q'} \bar{B}_{p'}$$

where the symbol objects $\bar{A}_q$ and $\bar{B}_{p'}$ indicate the active objects. Tailored for network applications, our model has the following features and differences:

- In this paper, we mostly consider the case when the communicating objects are of finite-states. However, when communicating objects in our model are augmented with some unbounded storage devices (such as a counter), it is difficult to directly translate transitions in such generalized service automata into P system rules. Hence, it is necessary to further study P systems on "automata objects" in addition to symbol and string objects.
- In P systems, the notion of "threads" or "processes" is hard to abstract. Naturally, in network service applications, such a notion is extremely important since, essen-

tially, the applications are distributed and concurrent in nature. Targeting at these applications, our model suggests a subclass of P systems where single/multiple processes can be clearly defined and, therefore, opens the door for further applying the model of P systems in areas like pervasive/mobile/distributed computing.
– In multiprocess service automata, we introduce the notion of regular maximal parallelism among processes, which is able to specify both Gh. Pãun's classical maximal parallelism and some other restricted forms of (maximal) parallelism [6, 10]. However, we shall point out that, for network applications, maximal parallelism in general is hard or expensive to implement. Therefore, it is a future research topic to study the cost of implementing restricted forms of regular maximal parallelism.

There has been much work on modeling distributed systems using automata. For instance, an input/output (I/O) automaton [12] models and reasons a concurrent and distributed discrete event system based on the broadcasting communication. The name "service automata" also appears in the work [11] that analyzes the behaviors over an open workflow nets. We reuse the name "service automata" in our paper but with completely different context and meaning. In short, in the aforementioned papers, a system is composed of a finite and fixed number of automata, while in our work, a service automaton runs on a multiset of automata (whose size is not specified when there is no initial constraint). The differences remain when one compares our work with some research in pervasive computing models [1–3] and mobile agents [16]. Linda [5] is another model of communications among processes, where communications are achieved by creating new objects in a tuple space, which is a quite practical model.

Our previous work, Bond Computing Systems [21], is also an addressless model to analyze network behaviors. However, the work treats a network system from a global view and focuses on how symbol objects (without states) are formed, without using maximal parallelism, into bonds, while in this paper we focus on automata objects and, from a local view, study processes on how state changes between objects.

## 2 Definitions

Let $\Sigma = \{A_1, ..., A_k\}$ ($k \geq 1$) be an alphabet of symbols. Each $A_i$, $i = 1, \cdots, k$, is called a *type*. An instance of a symbol $A_i$, for some $i$, in $\Sigma$ is called an *object* of type $A_i$, or simply an $A_i$-object. Without loss of generality, we call $A_1$ to be the *initial type*.

Each $A_i$ is associated with a (nondeterministic) finite automaton (we still use $A_i$ to denote it), which is a 3-tuple

$$A_i = (\mathcal{S}_i, \delta_i, q_{i0}),$$

where $\mathcal{S}_i = \{S_{i1}, ..., S_{il}\}$ (some $l \geq 1$) is a finite set of *internal states* (one can assume that the $\mathcal{S}_i$'s are disjoint), $\delta_i \subseteq \mathcal{S}_i \times \mathcal{S}_i$ is the set of the *internal* state transitions, and $q_{i0} \in \mathcal{S}_i$ is the initial state of the automaton $A_i$. We use $t_i : S_{iu} \rightarrow S_{iv}$ to denote a transition $t_i = (S_{iu}, S_{iv}) \in \delta_i$. In this way, an $A_i$-object itself is simply an instance of the finite automaton $A_i$.

Inter-object communications are achieved by *external* transitions in a given $\Delta$, and each external transition $r \in \Delta$ is in the following rule-form:

$$r : (A_i, t_i) \rightarrow (A_j, t_j),$$

for some $i$ and $j$, where $t_i \in \delta_i$ and $t_j \in \delta_j$ are internal state transitions. We will see in a moment that it is required that $t_i$ and $t_j$ associated with the $r$ must be fired together with $r$, and can not be fired alone. If an internal state transition $t$ is not associated with any external transition, we call such a $t$ as a *purely* internal state transition.

In summary, a *service automaton* is a tuple

$$G = \langle \Sigma, \Delta \rangle$$

where $\Sigma$ and $\Delta$ are specified in above. As we will see in a moment, $G$ could run over any number of objects and the number is not specified in $G$ itself when there is no initial constraint.

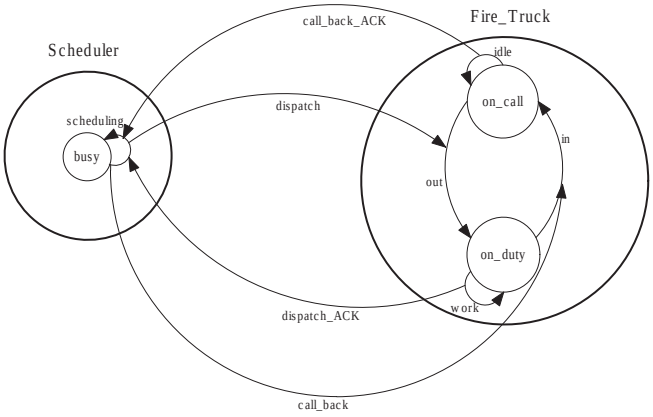


**Fig. 1.** An example service automaton $G$ for a fire truck scheduling system.

*Example 1.* Now we model a fire truck scheduling system by a service automaton $G = \langle \Sigma, \Delta \rangle$, where $\Sigma = \{Scheduler, Fire\_Truck\}$ with $Scheduler$ being the initial type, and $\Delta$ will be given in a moment. The service automaton $G$ is shown in Fig. 1, where automata $Scheduler$ and $Fire\_Truck$ are represented by the big (and bold) circles, internal state transitions (totally five) are represented by arrows within a big circle, and external transitions (totally four) are represented by arrows crossing the big circles. In $Scheduler$, $busy$ is the initial state, while in $Fire\_Truck$, $on\_call$ is the initial state. Again, the number of $Scheduler$-objects and $Fire\_Truck$-objects is not specified in $G$. □

We now define the semantics of the $G$. To specify an object $O$, we need only know its (unique) type $A$ and its (unique) current state $s$ of the finite automaton that is associated with the type; i.e., the $O$ is an instance of $(A, s)$, where for some $i$, $A = A_i \in \Sigma$ and $s \in S_i$ (sometimes, we just call $O$ an $(A, s)$ object).

A *collection* $(\mathcal{C}, O)$ is a multiset $\mathcal{C}$ of objects with $O \in \mathcal{C}$ being the only *active* object.

Let $(\mathcal{C}, O)$ and $(\mathcal{C}', O')$ be two collections and $r$ be a transition. We use

$$(\mathcal{C}, O) \xrightarrow{r} (\mathcal{C}', O')$$

to denote the fact that the collection $(\mathcal{C}, O)$ changes to the collection $(\mathcal{C}', O')$ by firing the transition $r$, which is defined formally as follows.

We first consider the case when $r$ is a purely internal transition, say $t_i : S_{iu} \rightarrow S_{iv}$ in $\delta_i$ (i.e., the transition is inside an $A_i$-object specified by the automaton $A_i$). We say that $O \xrightarrow{t_i} O'$ when $O$ is at state $S_{iu}$ and is of type $A_i$, and $O'$ is the result of changing the current state in $O$ with $S_{iv}$. Now,

$$(\mathcal{C}, O) \xrightarrow{r} (\mathcal{C}', O')$$

if the following conditions are satisfied:

- $O \xrightarrow{t_i} O'$.
- $\mathcal{C}'$ is the same as $\mathcal{C}$ except that the object $O$ is changed into $O'$.

Therefore, when the purely internal transition $t_i$ is fired, the active object must be at state $S_{iu}$ and, after firing the transition, the current state of the object is $S_{iv}$ and it remains as the active object.

Next, we consider the case when $r$ is an external transition, say $r : (A_i, t_i) \rightarrow (A_j, t_j)$, where $t_i : S_{iu} \rightarrow S_{iv}$ in $\delta_i$ and $t_j : S_{jp} \rightarrow S_{jq}$ in $\delta_j$ are internal state transitions. In this case,

$$(\mathcal{C}, O) \xrightarrow{r} (\mathcal{C}', O')$$

if, for some $O'' \in \mathcal{C}$ (with $O''$ and $O$ being distinct objects), and some object $O'''$,

- $O \xrightarrow{t_i} O'''$,
- $O'' \xrightarrow{t_j} O'$, and
- $\mathcal{C}'$ is the result of, in $\mathcal{C}$, replacing $O$ with $O'''$ and replacing $O''$ with $O'$.

Therefore, when the external transition $r$ is fired, the active object $O$ must be an $A_i$-object in state $S_{iu}$ and an $A_j$-object $O''$ in state $S_{jp}$ is nondeterministically chosen from the collection. The $A_i$-object $O$ will transit from state $S_{iu}$ to $S_{iv}$ (and evolve into $O'''$ defined in above), and the $A_j$-object $O''$ will transit from state $S_{jp}$ to $S_{jq}$ (and evolve into $O'$ defined in above), in parallel. After the transition is fired, the active object is changed from $O$ to $O'$.

Sometimes, it is useful to let a service automaton only run from a collection that satisfies an *initial constraint*. In such a constraint, one can designate certain types and explicitly require the number of objects for each such type to be a fixed constant, while for the remaining types, such numbers are not specified. An example initial constraint can be found in the scenario where one would like to specify a network service with an unspecified number of *client* type objects but with exactly one *central server* type object.

The collection $(\mathcal{C}, O)$ is *initial* if all objects in $\mathcal{C}$ are in their initial states, the $O$ is a designated initial and active object (i.e., the type of $O$ is the initial type $A_1$). and

the $\mathcal{C}$ satisfies an initial constraint. Unless stated otherwise, we implicitly assume that a service automaton is associated with an initial constraint.

For an initial collection $(\mathcal{C}, O)$, we write

$$(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O') \tag{1}$$

if there are collections $(\mathcal{C}, O) = (\mathcal{C}_0, O_0), \cdots, (\mathcal{C}_z, O_z) = (\mathcal{C}', O')$, for some $z$, such that

$$(\mathcal{C}_0, O_0) \stackrel{r_1}{\rightarrow} (\mathcal{C}_1, O_1) \cdots \stackrel{r_z}{\rightarrow} (\mathcal{C}_z, O_z), \tag{2}$$

for some (purely internal and external) transitions $r_1, \cdots, r_z$ in $G$.

In fact, $G$ defines a computing model that modifies a collection $(\mathcal{C}, O)$ into another collection $(\mathcal{C}', O')$ through $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$. To characterize the relationship $\rightsquigarrow_G$ that the $G$ can compute, we need more definitions.

Consider a set $T \subseteq \{(A_i, s)$, for all $s \in \mathcal{S}_i$, and for all $i\}$. For each pair $t = (A, s) \in T$, we use $\#_t(\mathcal{C}, O)$ to denote the number of the objects in $\mathcal{C}$ such that, each of which is of type $A$ and at state $s$. Clearly, when a proper ordering is applied on $T$, we may collect the numbers $\#_t(\mathcal{C}, O)$, $t \in T$, into a vector called $\#_T(\mathcal{C}, O)$. We use $R_{G,T}$, called the binary reachability of $G$ wrt $T$, to denote the set of all vector pairs $(\#_T(\mathcal{C}, O), \#_T(\mathcal{C}', O'))$ for all initial collections $(\mathcal{C}, O)$ and collections $(\mathcal{C}', O')$ satisfying $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$. In particular, when $T = \{(A_i, s)$, for all $s \in \mathcal{S}_i$, and for all $i\}$, we simply use $R_G$ to denote the $R_{G,T}$.

*Example 2.* We now explain the semantics of the example service automaton in Fig. 1. Roughly speaking, what the service automaton $G$ specifies is a fire truck scheduling system, where there could be an unspecified number of schedulers and fire trucks. Schedulers dispatch or call back fire trucks as needed, and once a fire truck changes its state, it sends back an acknowledge message to a scheduler. According to the finite automaton $Scheduler$, a scheduler is busy all the time. For the finite automaton $Fire\_Truck$, the internal state transition $out$ means that a fire truck is sent out to extinguish a fire, $in$ means that a fire truck finishes its work and comes back, $idle$ means that a fire truck keeps being on-call, and $work$ means that a fire truck keeps working (being on-duty).

The external transition $dispatch$ sends an on-call fire truck to extinguish a fire; $dispatch\_ACK$ describes that a dispatched fire truck sends an acknowledge message to a scheduler (we assume that all schedulers can communicate with each other through an underlying network); $call\_back$ simply makes a scheduler call an on-duty fire truck back; similar to $dispatch\_ACK$, $call\_back\_ACK$ means that once an on-duty fire truck is called back and becomes on-call, it sends an acknowledge message named $call\_back\_ACK$ to a scheduler.

In this example, we do not have an initial constraint. That is, $G$ could run over any number of $Scheduler$-objects and $Fire\_Truck$-objects. □

In the next example, we illustrate a scenario and explain in details how the example service automaton runs.

*Example 3.* We now illustrate an example run of the service automaton $G$ specified in Fig. 1. The run, as shown in Fig. 2, is on two schedulers and three fire trucks. Icons of telephones are used to depict $Scheduler$'s which are always in state $busy$, and icons of fire trucks are used for $Fire\_Truck$'s, while a fire truck with a black dot on it denotes a $Fire\_Truck$ in state $on\_duty$, otherwise denotes a $Fire\_Truck$ in state $on\_call$.

Consider $T = \{t_1, t_2, t_3\}$, where $t_1 = (Scheduler, busy)$, $t_2 = (Fire\_Truck, on\_call)$ and $t_3 = (Fire\_Truck, on\_duty)$. By definition, $\#_{t_1}(\mathcal{C}, O)$, $\#_{t_2}(\mathcal{C}, O)$, and $\#_{t_3}(\mathcal{C}, O)$ are the numbers of $Scheduler$'s in state $busy$, $Fire\_Truck$'s in state $on\_call$, and $Fire\_Truck$'s in state $on\_duty$ in a given collection $(\mathcal{C}, O)$, respectively. Let $\#_T(\mathcal{C}, O)$ be the vector $(\#_{t_1}(\mathcal{C}, O), \#_{t_2}(\mathcal{C}, O), \#_{t_3}(\mathcal{C}, O))$. We focus on an initial collection $(\mathcal{C}_0, O_0)$ where $O_0$ is an initial and active object (described in a moment), and, accordingly, $\#_T(\mathcal{C}_0, O_0) = (m, n, 0)$, where $m$ and $n$ could be any number. In this example, we assign $m = 2$ and $n = 3$.

Initially, according to the definition, there are only two $Scheduler$'s in state $busy$ and three $Fire\_Truck$'s in state $on\_call$, since $busy$ and $on\_call$ are the initial state of the automata $Scheduler$ and $Fire\_Truck$, respectively. Since $Scheduler$ is the initial type, we nondeterministically choose a $Scheduler$ in state $busy$, say $O_0$, as the initial and active object. Note that $O_0$ (the same for $O'_1, \cdots, O'_5, O_1, \cdots, O_5, O_6$ defined later) is only for notational convenience and it is not an identifier; actually, our system is addressless. Since all the internal state transitions are associated with external transitions, the internal state transitions cannot be fired alone, and hence, we only need to consider external transitions. According to Fig. 1, the external transition $dispatch$ requires some active $Scheduler$ in state $busy$ and some $Fire\_Truck$ in state $on\_call$; the external transition $dispatch\_ACK$ requires some active $Fire\_Truck$ in state $on\_duty$ and some $Scheduler$ in state $busy$; the external transition $call\_back$ requires some active $Scheduler$ in state $busy$ and some $Fire\_Truck$ in state $on\_duty$; and, finally, the external transition $call\_back\_ACK$ requires some active $Fire\_Truck$ in state $on\_call$ and some $Scheduler$ in state $busy$.

Initially, $dispatch$ is the only external transition that could be fired, since there are only two $Scheduler$'s in state $busy$ and three $Fire\_Truck$'s in state $on\_call$ in the initial collection, and the active object is some $Scheduler$ $O_0$. We nondeterministically pick a $Fire\_Truck$ in state $on\_call$, say $O'_1$, to fire $dispatch$. After firing $dispatch$, $O_0$ is still in state $busy$, while $O'_1$ changes to state $on\_duty$ (a black dot is added to $O'_1$ in Fig. 2 (1) to reflect the $on\_duty$ state), and becomes the active object $O_1$. Now, we have $(\mathcal{C}_0, O_0) \stackrel{dispatch}{\rightarrow} (\mathcal{C}_1, O_1)$ with $\#_T(\mathcal{C}_1, O_1) = (2, 2, 1)$. At this moment, the only firable external transition is $dispatch\_ACK$, which requires some active $Fire\_Truck$ in state $on\_duty$ and some $Scheduler$ in state $busy$. The active $Fire\_Truck$ in state $on\_duty$ is $O_1$, and we nondeterministically pick a $Scheduler$ in state $busy$, say $O'_2$. Note that $O'_2$ and $O_0$ may or may not (actually this is the case here) be the same object. After firing $dispatch\_ACK$, $O_1$ is still at state $on\_duty$, and $O'_2$ is still in state $busy$ and becomes active object $O_2$. So, we have $(\mathcal{C}_1, O_1) \stackrel{dispatch\_ACK}{\rightarrow} (\mathcal{C}_2, O_2)$, where $\#_T(\mathcal{C}_2, O_2) = (2, 2, 1)$. Fig. 2 (1) shows the run $(\mathcal{C}_0, O_0) \stackrel{dispatch}{\rightarrow} (\mathcal{C}_1, O_1) \stackrel{dispatch\_ACK}{\rightarrow} (\mathcal{C}_2, O_2)$.

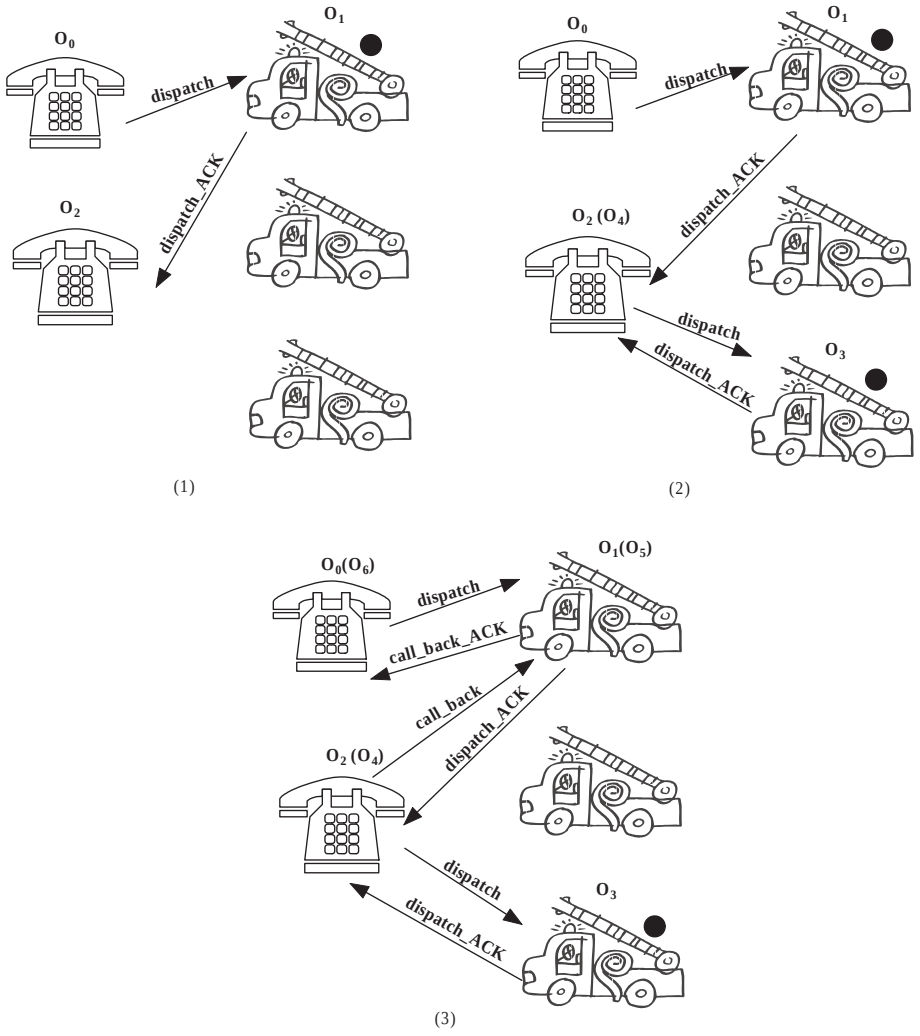


**Fig. 2.** An illustration of how the example service automaton $G$ in Fig. 1 runs on two schedulers and three fire trucks. Icons of telephones here denote $Scheduler$'s which are always in state $busy$, and icons of fire trucks here denote $Fire\_Truck$'s, while a fire truck with a black dot on it denotes a $Fire\_Truck$ in state $on\_duty$, otherwise denotes a $Fire\_Truck$ in state $on\_call$.

Next, both $dispatch$ and $call\_back$ become firable. Suppose that $dispatch$ is non-deterministically picked to fire, similarly, we get $(\mathcal{C}_2, O_2) \overset{dispatch}{\rightarrow} (\mathcal{C}_3, O_3)$ for some $Fire\_Truck\ O_3$ in state $on\_duty$ (a black dot is added on $O_3$ in Fig. 2 (2)), and $\#_T(\mathcal{C}_3, O_3) = (2, 1, 2)$. Next, $dispatch\_ACK$ becomes the only firable external transition again. Suppose that $(\mathcal{C}_3, O_3) \overset{dispatch\_ACK}{\rightarrow} (\mathcal{C}_4, O_4)$ for some $Scheduler\ O_4$ in

state $busy$ and $\#_T(\mathcal{C}_4, O_4) = (2, 1, 2)$. Fig. 2 (2) shows the run of $(\mathcal{C}_0, O_0) \overset{dispatch}{\rightarrow} (\mathcal{C}_1, O_1) \overset{dispatch\_ACK}{\rightarrow} (\mathcal{C}_2, O_2) \overset{dispatch}{\rightarrow} (\mathcal{C}_3, O_3) \overset{dispatch\_ACK}{\rightarrow} (\mathcal{C}_4, O_4)$.

Now, both $dispatch$ and $call\_back$ become firable. Suppose that this time $call\_back$ is nondeterministically picked to fire. We nondeterministically pick a $Fire\_Truck$ in state $on\_duty$ from $O_1$ and $O_3$ (in Fig. 2 (3), $O_1$ is picked), say $O'_5$, to fire $call\_back$. After firing $call\_back$, $O'_5$ changes to state $on\_call$ (the black dot is removed from $O'_5$ in Fig. 2 (3)) and becomes the active object $O_5$. We get $(\mathcal{C}_4, O_4) \overset{call\_back}{\rightarrow} (\mathcal{C}_5, O_5)$, where $\#_T(\mathcal{C}_5, O_5) = (2, 2, 1)$. Similarly, $call\_back\_ACK$ is the only firable external transition now, and we can get $(\mathcal{C}_5, O_5) \overset{call\_back\_ACK}{\rightarrow} (\mathcal{C}_6, O_6)$, for some $Scheduler\ O_6$ in state $busy$,and $\#_T(\mathcal{C}_6, O_6) = (2, 2, 1)$. Fig. 2 (3) shows the run $(\mathcal{C}_0, O_0) \overset{dispatch}{\rightarrow} (\mathcal{C}_1, O_1) \overset{dispatch\_ACK}{\rightarrow} (\mathcal{C}_2, O_2) \overset{dispatch}{\rightarrow} (\mathcal{C}_3, O_3) \overset{dispatch\_ACK}{\rightarrow} (\mathcal{C}_4, O_4) \overset{call\_back}{\rightarrow} (\mathcal{C}_5, O_5) \overset{call\_back\_ACK}{\rightarrow} (\mathcal{C}_6, O_6)$. Hence,

$$(\mathcal{C}_0, O_0) \rightsquigarrow_G (\mathcal{C}_6, O_6).$$

□

## 3 Decidability of Presburger Reachability

Let $Y = \{y_1, \cdots, y_m\}$ be a finite set of variables over integers. For all integers $a_y$, with $y \in Y$, $b$ and $c$ (with $b > 0$), $\sum_{y \in Y} a_y y < c$ is an *atomic linear relation* on $Y$ and $\sum_{y \in Y} a_y y \equiv_b c$ is a *linear congruence* on $Y$. A *linear relation* on $Y$ is a Boolean combination (using $\neg$ and $\wedge$) of atomic linear relations on $Y$. A *Presburger formula* $P(y_1, \cdots, y_m)$ [9] on $Y$ is a Boolean combination of atomic linear relations on $Y$ and linear congruences on $Y$. We say a vector $(z_1, \cdots, z_m)$ satisfies $P$ if $P(z_1, \cdots, z_m)$ holds.

A simple but important class of verification queries is about *reachability*. In this section, we study the Presburger reachability problem for service automata. Intuitively, the problem addresses whether there is a collection satisfying a given Presburger constraint is reachable. More precisely, the Presburger reachability problem is defined as follows:

Given: a service automaton $G$, a $T \subseteq \Sigma \times \mathcal{S}$, and a Presburger formula $\mathcal{P}$.

Question: Is there any initial collection $(\mathcal{C}, O)$ and some collection $(\mathcal{C}', O')$ such that $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$, and $\#_T(\mathcal{C}', O')$ satisfying $\mathcal{P}$?

In practice, the Presburger formula in the Presburger reachability problem is to specify some undesired property of a service automaton. Therefore, a positive answer to the reachability problem indicates that a network service modeled by the service automaton is not correctly designed.

Before we proceed further, we need more definitions. An $n$-dimensional *vector addition system with states* (VASS) $M$ is a 5-tuple $\langle V, p_0, p_f, S, \delta \rangle$ where $V$ is a finite set of *addition vectors* in $\mathbb{Z}^n$, $S$ is a finite set of *states*, $\delta \subseteq S \times S \times V$ is the *transition relation*, and $p_0, p_f \in S$ are the *initial state* and the final state, respectively. Elements $(p, q, v)$ of $\delta$ are called *transitions* and are usually written as $p \rightarrow (q, v)$.

A *configuration* of a VASS is a pair $(p, u)$ where $p \in S$ and $u \in \mathbb{N}^n$. The transition $p \rightarrow (q, v)$ can be applied to the configuration $(p, u)$ and yields the configuration $(q, u + v)$, provided that $u + v \geq \mathbf{0}$ (in this case, we write $(p, u) \rightarrow (q, u + v)$). For vectors $x$ and $y$ in $\mathbb{N}^n$, we say that $x$ can *reach* $y$, written $x \rightsquigarrow_M y$, if for some $j$, $(p_0, x) \rightarrow (p_1, x + v_1) \rightarrow \cdots \rightarrow (p_j, x + v_1 + ... + v_j)$ where $p_0$ is the initial state, $p_j$ is the final state, $y = x + v_1 + ... + v_j$, and each $v_i \in V$. It is well-known that Petri nets and VASS are equivalent. Consider a number $k \leq n$. We use $x(k)$ to denote the result of projecting the $n$-ary vector $x$ on its first $k$ components. An initial constraint $I$ is to restrict certain components in a vector $x$ to be fixed constants. We use $R^I_M(k)$ to denote all the pairs $(x(k), y(k))$ with $x \rightsquigarrow_M y$, and the starting vector $x$ satisfying $I$. When $k = n$, we simply write $R^I_M$ for $R^I_M(k)$. When $I$ is simply $true$, we write $R_M$ for $R^I_M$. We say that a service automaton $G$ can be simulated by a VASS $M$ if for some number $k$ and initial constraint $I$, $R_G = R^I_M(k)$. We say that a VASS $M$ can be simulated by a service automaton $G$ if for some $T$, $R_{G,T} = R_M$. If both ways are true, we simply say that they are equivalent (in terms of computing power).

**Theorem 1.** *Service automata are equivalent to VASS, and therefore the Presburger reachability problem of service automata is decidable.*

*Proof.* The proof consists of two parts. First, we prove that the service automaton can simulate an $n$-dimensional VASS $M$.

It is well known that VASS with no state are equivalent to VASS (with many states), and therefore, we assume that $M$ is specified by $m$ addition vectors

$$v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n}),$$

with $1 \leq i \leq m$, and the VASS is $n$-dimensional.

Now we describe how to construct a service automaton to simulate the VASS $M$. To make the description more concrete and easy to understand, we first give the service automaton that simulates the addition vector $(-3, +2)$ in Fig. 3.

We have three kinds of objects in the service automaton, namely, *counter objects*, *control objects*, and *addition vector objects*. Each object is associated with internal states.

Since the VASS $M$ is $n$-dimensional, we have totally $n$ types of counter objects, namely, $i$-th type counter objects, for $1 \leq i \leq n$. For each $i$-th type counter object, $CO_i$, its internal states are $CS_{i,1}$, $CS_{i,2}$, and $CS_{i,3}$. Recall that when the VASS $M$ runs, it changes the value of its configuration (which is simply a vector, called the configuration vector). The number of $i$-th type counter objects in state $CS_{i,2}$ represents the current value of the $i$-th component of the configuration vector in the VASS $M$. In Fig. 3, there are two types of counter objects, namely $CO_1$ and $CO_2$.

We also have a *control object* in the system. For the whole system, there is only one *control object* (recall that, this constitutes part of the initial constraint of the service automaton). This object has two states, namely $Control\_State1$ and $Control\_State2$, with $Control\_State1$ as its initial state.

For each addition vector, $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$, we have a corresponding object type $AO_i$. For each type $AO_i$, there is exactly one *addition vector object* in the whole

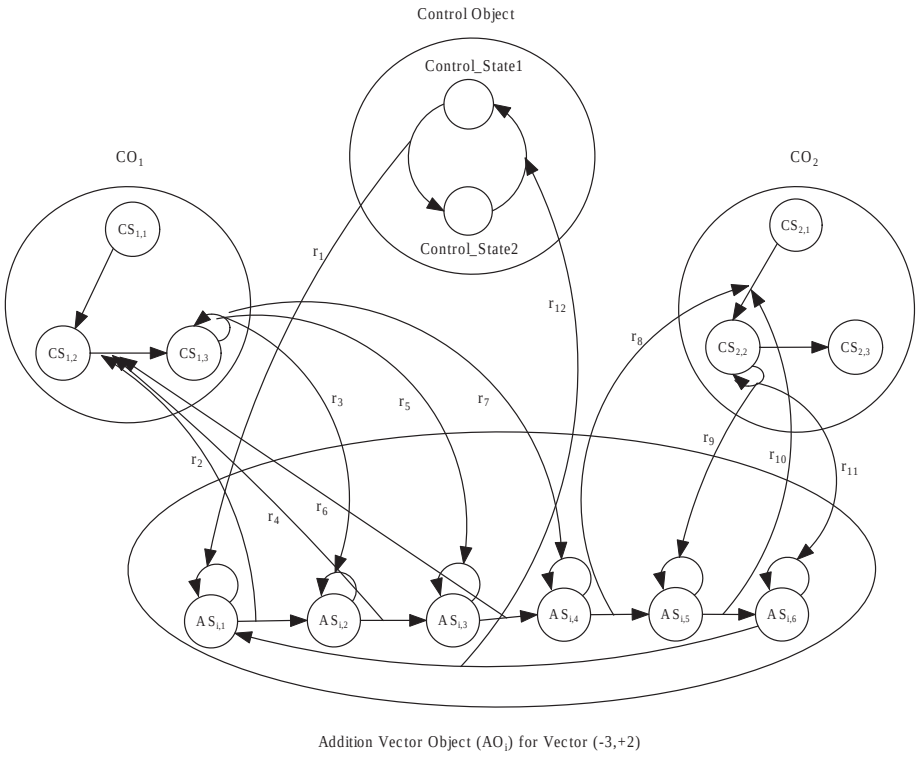


**Fig. 3.** Simulation of VASS using a service automaton

system. We use $|v_{i,j}|$ to denote the absolute value of $v_{i,j}$. The number of internal states of $AO_i$ is

$$l = \left( \sum_{j=1}^{n} |v_{i,j}| \right) + 1.$$

In Fig. 3, we have six internal states for the addition vector object that represents the addition vector $(-3, +2)$.

Internal state transitions and external transitions are demonstrated in Fig. 3.

Initially, the control object is the active object in the service automaton. Suppose $M$ fires its $i$-th addition vector, say, $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$. The $i$-th addition vector is simulated by a (rather long) sequence of transitions in the service automaton. Such simulation is shown in Fig. 3. As mentioned before, here we use $v_i = (-3, +2)$ (with $n = 2$). Initially, the control object fires the external transition $r_1$ which changes the control object's state from $Control\_State1$ to $Control\_State2$, and $r_1$, at the same time, also triggers the state change of the addition vector object of type $AO_i$. More precisely, the $AO_i$-object (from now on, we simply call it $AO_i$) issues a transition from $AS_{i,1}$ to $AS_{i,1}$. Now the active object is $AO_i$ and at state $AS_{i,1}$. Then $AO_i$ will execute

some external transition. Depending on whether $v_{i,j}$ is positive or not, the constructions of external transition rules are different.

If $v_{i,j}$ is negative (which is the case here, since $v_{i,1} = -3$), this means we need to decrease the number of $CO_j$-objects (here is the $CO_1$-objects) in state $CS_{j,2}$ (here is $CS_{1,2}$), hence $r_2$ is fired. It will change its state from $AS_{i,1}$ to $AS_{i,2}$. At the same time, $r_2$ will make the counter object, $CO_1$, change the state. Thus the $CO_1$-object will change the state from $CS_{1,2}$ to $CS_{1,3}$. This accomplishes one subtraction. Now the $CO_1$-object becomes the active object. To let the process to continue, the $CO_1$-object executes $r_3$ to transfer the active object back to $AO_i$. It will change its own state to change from $CS_{1,3}$ to $CS_{1,3}$, and $AO_i$ will change the state from $AS_{i,2}$ to $AS_{i,2}$. Now $AO_i$ is the active object. Next the process executes $r_4$, $r_5$, $r_6$, and $r_7$ sequentially, resulting that the number of $(CO_1, CS_{1,2})$ is subtracted by 3 (which is corresponding to $v_{i,1} = 3$ ), and $AO_i$ is still the active object.

If $v_{i,j}$ is positive (which is the case of $v_{i,2} = 2$), this means we need to increase the number of $CO_j$-objects (here is the $CO_2$-objects) in state $CS_{j,2}$ (here is $CS_{2,2}$), hence next $r_8$ is fired. Thus the $CO_2$ object will change the state from $CS_{2,1}$ to $CS_{2,2}$. This accomplishes one addition. Now $CO_2$ becomes the current active object. Next the process executes $r_9$, $r_{10}$, and $r_{11}$ sequentially, resulting that the number of $(CO_2, CS_{2,2})$ is added by 2 (which is corresponding to $v_{i,2} = 2$ ), and $AO_i$ is still the active object.

To make this description general, let's assume $AO_i$ is at the state $AS_{i,j}$, where $j = \left(\sum_{h=1}^{k} |v_{i,h}|\right) + l$ with $1 \leq l \leq v_{i,k+1}$. To accomplish the operation of $v_{i,k+1}$, there are still $(|v_{i,k+1}| - l)$ more steps to go.

$AO_i$ will execute an external transition. It will change its state from $AS_{i,j}$ to $AS_{i,j+1}$. At the same time, this external transition will make the counter object, $CO_{k+1}$, change the state. Depending on whether $v_{i,k+1}$ is positive or not, there are two cases.

If $v_{i,k+1}$ is negative, this means we need to decrease the number of $CO_{k+1}, CS_{k+1,2}$ objects. Thus the $CO_{k+1}$ object will change the state from $CS_{k+1,2}$ to $CS_{k+1,3}$. This accomplishes one subtraction. Now $CO_{k+1}$ becomes the current active object. For the process to continue, it needs to transfer the active role back to $AO_i$. Then, $CO_{k+1}$ will execute an external transition. It will change its state to change from $CS_{k+1,3}$ to $CS_{k+1,3}$. $AO_i$ will change the state from $AS_{i,j+1}$ to $AS_{i,j+1}$. Now $AO_i$ will become the current active object and its state is in $AS_{i,j+1}$.

If $v_{i,k+1}$ is positive, this means we need to increase the number of $CO_{k+1}$-objects in state $CS_{k+1,2}$. Thus the $CO_{k+1}$-object will change the state from $CS_{k+1,1}$ to $CS_{k+1,2}$. This accomplishes one addition. Now $CO_{k+1}$ becomes the current active object. For the process to continue, it needs to transfer the active role back to the $AO_i$. Then, $CO_{k+1}$ will execute an external transition. It will change its state to change from $CS_{k+1,2}$ to $CS_{k+1,2}$. $AO_i$ will change the state from $AS_{i,j+1}$ to $AS_{i,j+1}$. Now the $AO_i$ will become the current active object and its state is in $AS_{i,j+1}$.

As the above process continues, at the point that $AO_i$ is active and at the state $AS_{i,z}$, $z = \left(\sum_{j=1}^{n} |v_{i,j}|\right) + 1$, and $AO_i$ has performed the internal transition from $AS_{i,z}$ to $AS_{i,z}$, all the operations of $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$ have been finished. Now we need to make the control object to be active to fire a next addition vector. Thus $AO_i$ fires an external transition where $AO_i$ changes its state from $AS_{i,z}$ to $AS_{i,1}$. At the same time,

the control object changes its state from $Control\_State2$ to $Control\_State1$. Thus the control object becomes the active object again. In Fig. 3, this external transition is labeled by $r_{12}$. The addition vector $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$'s function is accomplished. The control object can continue to simulate firing a next addition vector.

It is not hard to see that the service automaton will simulate $M$.

For the second part of the proof, we need show that a VASS $M$ can simulate a service automaton $G$. Suppose that there are $m$ types of objects in $G$, and without loss of generality, we assume that internal states in different types of objects are all distinct. Therefore, an internal state of an object can uniquely tell the type of the object (hence, we do not need to refer an object type in an internal transition and an external transition, in below). Suppose that all the internal states, $S$, are properly ordered (we use $position(s)$ to indicate the position of state $s$ in the ordering), and we use $\#_s$ to indicate the current number of objects in state $s$ at some moment when the service automaton runs. We use $\#$ to indicate the array of all the $\#_s$. The VASS $M$ constructed in below is to update the vector $\#$ while transitions in $G$ is executed.

In $M$, the states are exactly those in $S$, the internal states of objects in $G$.

For each internal transition, say $s \rightarrow s'$, we have an addition vector along with a state transition in $M$ as follows: $\langle s, (0, \cdots, 0, -1, 0 \cdots, 0, +1, 0, \cdots, 0), s' \rangle$, where the $-1$ is at position $position(s)$ and the $+1$ is at position $position(s')$. The vector corresponds to the fact that, after firing the internal transition, there is one object in state $s$ evolving into an object in state $s'$. The state transition (from $s$ to $s'$) in $M$ corresponds to the fact that, after firing the internal transition, the active object is transferred from an object at state $s$ to an object at state $s'$.

For each external transition, say $(s_1, s_2) \rightarrow (s_3, s_4)$ (where $(s_1, s_2)$ and $(s_3, s_4)$ represent two internal transitions), we have an addition vector along with a state transition in $M$ as follows:

$$\langle s_1, (0, \cdots, 0, -1, 0 \cdots, 0, +1, 0, \cdots, 0, -1, 0 \cdots, 0, +1, 0, \cdots, 0), s_4 \rangle,$$

where the two $-1$'s are at positions $position(s_1)$ and $position(s_3)$, respectively, and the two $+1$'s are at positions $position(s_2)$ and $position(s_4)$, respectively. The vector corresponds to the fact that, after firing the external transition, there is one object in state $s_1$ evolving into an object in state $s_2$, and at the same time, there is one object in state $s_3$ evolving into an object in state $s_4$. The state transition (from $s_1$ to $s_4$) in $M$ corresponds to the fact that, after firing the external transition, the active object is transferred from an object at state $s_1$ to an object at state $s_4$. Clearly, $M$ faithfully simulates $G$ (assuming that $M$ starts with vectors that encode the initial collections of $G$).

The two parts complete our proof.

□

The above theorem characterizes the computing power of service automata, when the service automata are interpreted as computation devices. In the following, we will treat service automata as language acceptors and therefore, we can characterize the processes that are generated by such services. We need more definitions.

Let $\Pi = \{a_1, \cdots, a_n\}$ ($n \geq 1$) be an alphabet of *(activity) labels*. Now, we are given a function that assigns each purely internal transition with $\Lambda$ (empty label) and assigns each external transition with either $\Lambda$ or an activity label in $\Pi$. Recall that we

write $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$ if there are collections $(\mathcal{C}_0, O_0), \cdots, (\mathcal{C}_z, O_z)$, for some $z$, such that

$$(\mathcal{C}_0, O_0) \xrightarrow{r_1} (\mathcal{C}_1, O_1) \cdots \xrightarrow{r_z} (\mathcal{C}_z, O_z),$$

for some purely internal and external transitions $r_1, \cdots, r_z$ in $G$. We use $\alpha$ to denote the sequence of labels for transitions $r_1, \cdots, r_z$. To emphasize the $\alpha$, we simple write $(\mathcal{C}, O) \rightsquigarrow_G^{\alpha} (\mathcal{C}', O')$ for $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$. In this case, we say that $\alpha$ is a *process* of $G$. The set $L(G)$ of all processes of $G$ is called *the service* defined by the service automaton $G$.

*Example 4.* Consider the example service automata in Fig. 1. We assign $dispatch$, $dispatch\_ACK$, $call\_back$ and $call\_back\_ACK$ with $a_1, \Lambda, a_2$ and $\Lambda$, respectively. The service, by definition, $L(G) = \{\alpha : (\mathcal{C}, O) \rightsquigarrow_G^{\alpha} (\mathcal{C}', O')\}$. Define $\#_a(w)$ as the number of symbol $a$ appearing in a word $w$. We can easily get that $L(G) = \{\alpha : \#_{a_1}(\alpha') \geq \#_{a_2}(\alpha')$ for all prefix $\alpha'$ of $\alpha\}$, since the number of fire trucks dispatched is always greater that the number fire trucks called back. Hence, the service $L(G)$ specified by the service automata in Fig. 1 is nonregular. □

A multicounter machine $M$ is a nondeterministic finite automaton (with one-way input tape) augmented with a number of counters. Each counter takes nonnegative integer values and can be incremented by 1, decremented by 1, and tested for 0. It is well known that when $M$ has two counters, it is universal. A counter is blind if it can not be tested for 0, however, when its value becomes negative, the machine crashes. A blind counter machine is a multicounter machine $M$ whose counters are blind and the counters become 0 when computation ends.

**Theorem 2.** *Services defined by service automata can be accepted by blind counter machines.*

The above result follows from the well-known fact that blind counter machines are essentially VASS treated as a language acceptor (with Presburger acceptance condition; i.e., the VASS stops when a pre-given Presburger formula is satisfied by the current vector). From that fact, it is not hard to show that service automata can define fairly complex processes, which are not necessarily regular, context free, or semilinear. Therefore, we are curious on what will happen if we put some restrictions over the syntax of service automata and what characteristics are essential to the computation power of service automata.

One interesting case is when a service automaton only has objects of one type; i.e., $\Sigma$ is of size 1, say, $\Sigma = \{A\}$. We call such a service automaton as a *1-type* service automaton. We get the following result, which implies that the number of object types is not critical to the computation power when there is no initial constraint. Currently, we do not know the precise relationship between 1-type service automata and service automata both with initial constraints.

**Theorem 3.** *1-type service automata without initial constraints can simulate any service automata without initial constraints, and, therefore, both define the same class of services.*

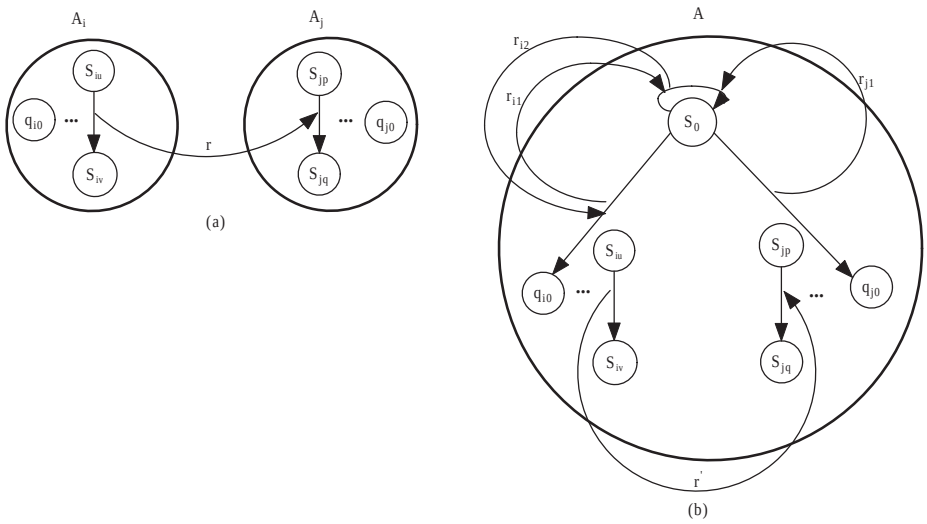


**Fig. 4.** (a) A service automaton $G$ (part). (b) A 1-type service automaton $G'$ to simulate G.

*Proof.* Given a service automaton $G = \langle \Sigma, \Delta \rangle$ as shown in Fig. 4. (a), we now construct a 1-type service automaton $G'$ to simulate $G$. In $G'$, we only have $A$-objects, and inside $A$, we define the initial state of $A$ as $S_0$, and faithfully copy all the states from $G$, while keep all the internal state transitions. For each $A_i$ in $G$, suppose $q_{i0}$ is the initial state, then in $G'$, we define the following external transition to produce $(A, q_{i0})$ objects:

$$r_{i1} : (A, t_{i1}) \rightarrow (A, t_0),$$

where $t_0 : S_0 \rightarrow S_0$ and $t_{i1} : S_0 \rightarrow q_{i0}$. In particular, each time when $r_{i1}$ is fired, $t_{i1}$ generates an $(A, q_{i0})$ object, and $t_0$ lets an $(A, S_0)$ object being active in order to fire some next transition. If $A_i$ is the intial type, we additionally define:

$$r_{i2} : (A, t_0) \rightarrow (A, t_{i1}),$$

where $t_0$ and $t_{i1}$ are the same as in $r_{i1}$. Once $r_{i2}$ is fired, an $(A, q_{i0})$ object will become active, and $r_{i1}$'s, for all $i$, cannot be fired any more, since there is no $(A, S_0)$ object available. We label $r_{i1}$ and $r_{i2}$ with $\Lambda$'s (empty labels). Now we are to simulate the external transitions of $G$. Suppose there is an arbitrary external transition $r \in \Delta$ defined as

$$r : (A_i, t_i) \rightarrow (A_j, t_j),$$

where $t_i : S_{iu} \rightarrow S_{iv}$ and $t_j : S_{jp} \rightarrow S_{jq}$ are internal state transitions. Corresponding to $r$, in $G'$ we have

$$r' : (A, t'_i) \rightarrow (A, t'_j),$$

where $t'_i : \ S_{iu} \rightarrow S_{iv}$ and $t'_j : \ S_{jp} \rightarrow S_{jq}$ are internal state transitions of $A$. Besides, we assign $r'$ with the same activity label as $r$. We leave it to readers to check that $L(G) = L(G')$.

□

Our next question focuses on whether purely internal state transitions are necessary for a service automaton. We call a service automaton without purely internal state transitions as *internal-free*.

**Theorem 4.** *Any service automaton can be simulated by an internal-free service automaton .*

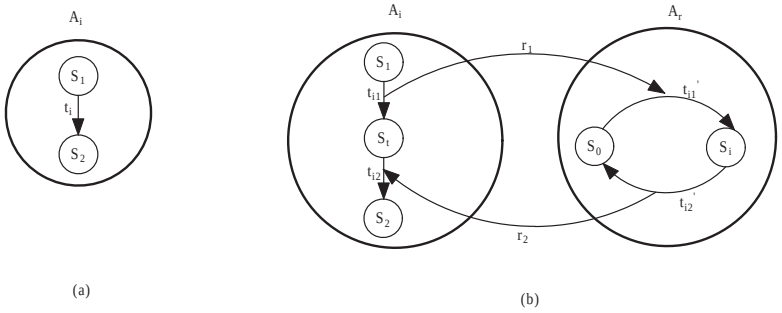


**Fig. 5.** (a) A service automaton $G$ with pure internal transition(s). (b) An internal-free service automaton $G'$ to simulate $G$.

*Proof.* Given a service automaton $G$, we construct an internal-free service automaton $G'$ to simulate it, as in Fig. 5. First, we copy the whole $G$ into $G'$, keep all the external transitions, and preserve the initial constraint. Plus, we have an auxiliary object $A_r$ in $G'$, with $S_0$ as its initial state. Consider an arbitrary pure internal state transition $t_i$ of $A_i$ in $G$, $t_i : \ S_1 \rightarrow S_2$. For the corresponding $A_i$ in $G'$, we add an intermediate state $S_t$, and split $t_i$ into $t_{i1}$ and $t_{i2}$, where $t_{i1} : \ S_1 \rightarrow S_t$ and $t_{i2} : \ S_t \rightarrow S_2$. For each $t_i$ in $G$, we have a corresponding state $S_i$ in $A_r$, and define $t'_{i1} : \ S_0 \rightarrow S_i$ and $t'_{i2} : \ S_i \rightarrow S_0$. We define external transitions $r_1$ and $r_2$ as:

$$r_1 : \ (A_i, t_{i1}) \rightarrow (A_r, t'_{i1}),$$

and

$$r_2 : \ (A_r, t'_{i2}) \rightarrow (A_i, t_{i2}).$$

Obviously, $r_1$ and $r_2$ together can faithfully simulate $t_i$, in the sense that $\#_T(\mathcal{C}, O)$ of $G'$ is the same as $\#_T(\mathcal{C}, O)$ of $G$, respect to the same $T$. Besides, after firing $r_1$ and $r_2$, the $A_r$-object is still in state $S_0$, which implies that $A_r$ is ready for the simulation of the next internal state transition. Hence $G'$ can consistently simulate $G$, and $R_{G,T} = R_{G',T}$, for any $T \subseteq \{(A_i, s), \text{ for all } A_i \in \Sigma, \text{ and for all } s \in \mathcal{S}_i\}$, where $\Sigma$ is the alphabet of symbols of $G$, and $\mathcal{S}$ is the set of all states of $G$.

□

However, currently, it is a difficult problem to characterize a nontrivial class of service automata that only define regular services.

## 4 Multiprocess Service Automata

In previous sections, we model a single process from the local view of the active object; i.e., we view a single process following the flow of its active object. We describe a process in a service automaton $G$ by recording the trace of the active object in a certain collection, as given in (2),

$$(\mathcal{C}_0, O_0) \xrightarrow{r_1} (\mathcal{C}_1, O_1) \cdots \xrightarrow{r_z} (\mathcal{C}_z, O_z),$$

for some purely internal and external transitions $r_1, \cdots, r_z$ in $G$.

In a real network, there are often multiple processes in execution at the same time, and each process has its own active object. To model multiprocess processes, we need to take all the active objects into consideration.

Let $G = \langle \Sigma, \Delta \rangle$ be a service automaton, and we can define a corresponding *multiprocess* service automaton $G_{\mathtt{mp}}$ as follows. A *multiprocess collection* $(\mathcal{C}, \mathcal{O})$ is a multiset $\mathcal{C}$ of objects with $\mathcal{O} \subseteq \mathcal{C}$ being the *active multiset* (i.e., each object in $\mathcal{O}$ is active). Suppose that there are totally $m$ purely internal and external transitions $r_1, \cdots, r_m$ in $G$. Let $R = \{r_1^{n_1}, \cdots, r_m^{n_m}\}$ be a *transition multiset* where each $n_i \in \mathbb{N} \cup \{*\}$ $(1 \leq i \leq m)$ is the multiplicity of transition $r_i$ (the meaning of $*$ will be made clear in a moment). A *multiprocess service automaton* is a tuple

$$G_{\mathtt{mp}} = \langle \Sigma, \Delta, \mathcal{R} \rangle,$$

where $\mathcal{R}$ is a finite set of transition multisets. For each transition multiset $R = \{r_1^{n_1}, \cdots, r_m^{n_m}\} \in \mathcal{R}$, we have a corresponding Presburger formula $P_R(y_1, \cdots, y_m)$ defined in this way: for each $i$, when $n_i \in \mathbb{N}$, we define an atomic linear relation $P_i$ as $y_i = n_i$; when $n_i = *$, $P_i$ is defined as $y_i \geq 0$. Finally, $P = \bigwedge_i P_i$. For instance, for the transition multiset $\{r_1^*, r_5^2\}$ (transitions with 0 multiplicity are omitted in the $R$), the corresponding Presburger formula $P_R(y_1, \cdots, y_m)$ is

$$y_1 \geq 0 \wedge y_5 = 2 \bigwedge_{i \neq 1,5} y_i = 0.$$

Let $(\mathcal{C}, \mathcal{O})$ and $(\mathcal{C}', \mathcal{O}')$ be two multiprocess collections, $R$ be a transition multiset in $\mathcal{R}$. Now,

$$(\mathcal{C}, \mathcal{O}) \xrightarrow{R} (\mathcal{C}', \mathcal{O}')$$

if the following conditions are satisfied:

(i) there are some *disjoint* multisets $\mathcal{C}_j \subset \mathcal{C}$, each of which satisfies the following: $(\mathcal{C}_j, O_j) \xrightarrow{r_i} (\mathcal{C}'_j, O'_j)$ for some transition $r_i$, multisets $\mathcal{C}'_j$, and objects $O_j$ and $O'_j$ (Notice that, by definition of $\xrightarrow{r_i}$, $O_j \in \mathcal{C}_j$ and $O'_j \in \mathcal{C}'_j$). Notice that, for each $j$, the $r_i$ is fired once.

(ii) suppose that the total number of times that transitions $r_i$ are fired in (i) is $\#_{r_i}$. Then, the corresponding Presburger formula $P_R(\#_{r_1}, \cdots, \#_{r_m})$ of $R$ holds.
(iii) the $r_i$'s are fired in a maximally parallel manner. That is, $r_i$ should be fired as many times as possible under (i) and (ii); i.e., the vector $(\#_{r_1}, \cdots, \#_{r_m})$ satisfying (i) and (ii) is maximal.
(iv) the $\mathcal{C}'$ is the result of replacing each sub-multiset $\mathcal{C}_j$ in $\mathcal{C}$ with $\mathcal{C}'_j$, and the $\mathcal{O}'$ is result of replacing each object $O_j$ in $\mathcal{O}$ with $O'_j$.

Actually, $(\mathcal{C}, \mathcal{O}) \xrightarrow{R} (\mathcal{C}', \mathcal{O}')$ fires transitions in $R$ in a maximally parallel manner with respect to the constraint $P_R$. By definition, the Presburger formula $P_R$ is only capable of comparing a variable with a constant. Hence, the maximally parallel notion used in here is called *regular maximal parallelism*. It is not hard to see that it generalizes Pãun's [14] classic maximal parallelism (taking the transition multiset in the form of $\{r_1^*, \cdots, r_m^*\}$) as well as some other restricted forms [6, 10].

Similar to service automata, one can associate an initial constraint to a multiprocess service automata. A multiprocess collection $(\mathcal{C}, \mathcal{O})$ is initial if the initial active multiset $\mathcal{O}$ are of the initial type $A_1$ in the initial state, and the collection $\mathcal{C}$ satisfies the given initial constraint. For an initial multiprocess collection $(\mathcal{C}, \mathcal{O})$, we write

$$(\mathcal{C}, \mathcal{O}) \rightsquigarrow_{G_{\mathtt{mp}}} (\mathcal{C}', \mathcal{O}') \tag{3}$$

if, for some $z$, there are multiprocess collections

$$(\mathcal{C}, \mathcal{O}) = (\mathcal{C}_0, \mathcal{O}_0), (\mathcal{C}_1, \mathcal{O}_1), \cdots, (\mathcal{C}_z, \mathcal{O}_z) = (\mathcal{C}', \mathcal{O}')$$

such that

$$(\mathcal{C}_0, \mathcal{O}_0) \xrightarrow{R_1} (\mathcal{C}_1, \mathcal{O}_1) \cdots \xrightarrow{R_z} (\mathcal{C}_z, \mathcal{O}_z), \tag{4}$$

for some transition multisets $R_1, \cdots, R_z$ in $\mathcal{R}$.

Similarly, we can define $\#_t(\mathcal{C}, \mathcal{O})$ for $t = (A, s)$ as the number of $(A, s)$ objects in $\mathcal{C}$, and the vector $\#_T(\mathcal{C}, \mathcal{O})$ as well as the binary reachability $R_{G_{\mathtt{mp}},T}$ can be defined similarly to single-process service automata.

*Example 5.* Example 3 gives a service automaton that models a fire truck scheduling system, where transitions are fired sequentially. In the real world, if there are multiple schedulers, they can work in parallel; i.e., some schedulers may dispatch on-call fire trucks, some schedulers may call back on-duty fire trucks, and those actions can happen in parallel, only if two different actions work upon disjoint objects. Based on this observation, we can define a multiprocess service automaton $G_{\mathtt{mp}} = \langle \Sigma, \Delta, \mathcal{R} \rangle$ based on the example service automaton $G$ defined in Example 1, and $\mathcal{R} = \{R\}$ where $R$ is the only transition multiset defined in below:

$$R = \{dispatch^*, dispatch\_ACK^*, call\_back^*, call\_back\_ACK^*\},$$

which means that $R$ fires the four transitions in a maximally parallel manner.

Suppose that $\mathcal{O}_0 = \{(Scheduler, busy)^5\}$ as the initial active set, i.e., initially, there are five $Scheduler$'s in state $busy$ which are ready to start five processes.

As Example 3, we define $T = \{t_1, t_2, t_3\}$, where $t_1 = (Scheduler, busy)$, $t_2 = (Fire\_Truck, on\_call)$ and $t_3 = (Fire\_Truck, on\_duty)$. We focus on the initial multiprocess collection $(\mathcal{C}_0, \mathcal{O}_0)$, where $\#_T(\mathcal{C}_0, O_0) = (m, n, 0)$, with $m \geq 5$ and $n$ can be any number. Let us designate $m = 5$ and $n = 8$; i.e., there are five $Scheduler$'s and eight $Fire\_Truck$'s in the initial multiprocess collection. In below, we will illustrate how the multiprocess service automaton runs.

Following the analysis in Example 3, we know that initially $dispatch$ is the only external transition that can be fired, and there are at most five $dispatch$'s that can be fired, since there are only five $Scheduler$'s. After firing $R$, we have five $Fire\_Truck$'s in state $on\_duty$ and they all become active, and there are (8-5=3) three $Fire\_Truck$'s in state $on\_call$. Now, $(\mathcal{C}_0, \mathcal{O}_0) \xrightarrow{R} (\mathcal{C}_1, \mathcal{O}_1)$ with $\mathcal{O}_1 = \{(Fire\_Truck, on\_duty)^5\}$ and $\#_T(\mathcal{C}_1, \mathcal{O}_1) = (5, 3, 5)$. Next, there are at most five $dispatch\_ACK$'s that can be fired, and $(\mathcal{C}_1, \mathcal{O}_1) \xrightarrow{R} (\mathcal{C}_2, \mathcal{O}_2)$, where $\mathcal{O}_2 = \{(Scheduler, busy)^5\}$ and $\#_T(\mathcal{C}_2, \mathcal{O}_2) = (5, 3, 5)$. At this moment, $R$ can fire both $dispath$ and $call\_back$, and in fact, we can fire five of them totally at most. Suppose that we nondeterministically pick one $dispath$ and four $call\_back$'s to fire, and then one new $Fire\_Truck$ is dispatched, and four on-duty $Fire\_Truck$'s are called back, and hence there are (3+4-1=6) six $Fire\_Truck$'s in state $on\_call$ with four of them active, and (5-4+1=2) two $Fire\_Truck$'s in state $on\_duty$ with one of them active. That is, we have $(\mathcal{C}_2, \mathcal{O}_2) \xrightarrow{R} (\mathcal{C}_3, \mathcal{O}_3)$, where $\mathcal{O}_3 = \{(Fire\_Truck,\ on\_duty)^1,\ (Fire\_Truck, on\_call)^4\}$ and $\#_T(\mathcal{C}_3, \mathcal{O}_3) = (5, 6, 2)$. At this time, there are at most one $dispath\_ACK$ and four $call\_back\_ACK$'s could be fired, and hence $(\mathcal{C}_3, \mathcal{O}_3) \xrightarrow{R} (\mathcal{C}_4, \mathcal{O}_4)$, where $\mathcal{O}_4 = \{(Scheduler, busy)^5\}$ and $\#_T(\mathcal{C}_4, \mathcal{O}_4) = (5, 6, 2)$. Therefore, we have

$$(\mathcal{C}_0, \mathcal{O}_0) \rightsquigarrow_{G_{\mathtt{mp}}} (\mathcal{C}_4, \mathcal{O}_4).$$

□

Obviously, multiprocess service automata can simulate service automata, and hence they can simulate VASS. Next we will show that multiprocess service automata are strictly more powerful than VASS.

## 5 Undecidability of Presburger Reachability for Multiprocess Service Automata

Now we study the Presburger reachability problem for multiprocess service automata:

Given: a multiprocess service automaton $G_{\mathtt{mp}}$, a set $T \subseteq \Sigma \times \mathcal{S}$, and a Presburger formula $\mathcal{P}$.

Question: Are there some initial multiprocess collection $(\mathcal{C}, \mathcal{O})$ and some multiprocess collection $(\mathcal{C}', \mathcal{O}')$ such that $(\mathcal{C}, \mathcal{O}) \rightsquigarrow_{G_{\mathtt{mp}}} (\mathcal{C}', \mathcal{O}')$ and $\#_T(\mathcal{C}', \mathcal{O}')$ satisfying $\mathcal{P}$?

To proceed further, we need more definitions. A *linear polynomial* over nonnegative integer variables $x_1, \cdots, x_n$ is a polynomial of the form $a_0 + a_1x_1 + ... + a_nx_n$ where each coefficient $a_i$, $0 \leq i \leq n$, is an integer. The polynomial is nonnegative if each coefficient $a_i$, $0 \leq i \leq n$ is in $\mathbb{N}$. A *k-system* is a quadratic Diophantine equation system

that consists of $k$ equations over nonnegative integer variables $x_1, \cdots, x_m, y_1, \cdots, y_n$ for some $m, n$, in the following form:

$$\begin{cases} \sum_{1 \le j \le l} B_{1j}(y_1, \cdots, y_n) A_{1j}(x_1, \cdots, x_m) = C_1(x_1, \cdots, x_m) \\ \qquad\qquad \vdots \\ \sum_{1 \le j \le l} B_{kj}(y_1, \cdots, y_n) A_{kj}(x_1, \cdots, x_m) = C_k(x_1, \cdots, x_m) \end{cases} \tag{5}$$

Where the $A$'s, $B$'s and $C$'s are nonnegative linear polynomials, and $l, m, n$ are positive integers.

[20] points out that the $k$-system in (5) can be simplified into the following form:

$$\begin{cases} y_1 A_{11}(x_1, \cdots, x_m) + \cdots + y_n A_{1n}(x_1, \cdots, x_m) = C_1(x_1, \cdots, x_m) \\ \qquad\qquad \vdots \\ y_1 A_{k1}(x_1, \cdots, x_m) + \cdots + y_n A_{kn}(x_1, \cdots, x_m) = C_k(x_1, \cdots, x_m) \end{cases} \tag{6}$$

**Theorem 5.** *If the Presburger reachability problem of multiprocess service automata is decidable, then it is decidable whether a k-system has a solution for any $k$.*

*Proof.* We will construct a multiprocess service automaton $G_{\mathtt{mp}}$ from a $k$-system specified in (6). In the $k$-system, an $\langle i, j \rangle$-term (resp. an $\langle i \rangle$-term) is to indicate the term $x_i y_j$ (resp. $x_i$), for $1 \le i \le m$ and $1 \le j \le n$. Clearly, the $k$-system can be expressed as a Presburger formula $\mathcal{P}$ over all of the $\langle i, j \rangle$-terms and the $\langle i \rangle$-terms.

$G_{\mathtt{mp}}$ is constructed as follows. It has only one object type, say $A$. Inside $A$, we have states $S_{\text{initial}}, S_{\text{warehouse}}, S_{\langle i,j \rangle}, S_{\langle i \rangle}$, for $1 \le i \le m$ and $1 \le j \le n$. Intuitively, the number of objects at state $S_{\langle i,j \rangle}$, denoted by $\#S_{\langle i,j \rangle}$, is to encode the $\langle i, j \rangle$-term $x_i y_j$. Similarly, the number of objects at state $S_{\langle i \rangle}$, denoted by $\#S_{\langle i \rangle}$, is to encode the $\langle i \rangle$-term $y_i$.

Fig. 6 shows part of $G_{\mathtt{mp}}$ for simulating terms $x_1 y_j, \cdots, x_m y_j$. The complete diagram of $G_{\mathtt{mp}}$ can be obtained by, respectively for each $j$, connecting the states $S_{\langle 1,j \rangle}, \cdots, S_{\langle m,j \rangle}$ to states $S_{\text{initial}}, S_{\text{warehouse}}, S_{\langle 1 \rangle}, \cdots, S_{\langle m \rangle}$ shown in the figure. The transitions $r$'s, $h$'s, $g$'s stated in below are all illustrated in the figure.

The automaton $G_{\mathtt{mp}}$ works as follows. Initially, all the objects are at the initial state $S_{\text{initial}}$, among which some are active. It first fires a transition multiset

$$R_{\text{initial}} = \{g^*_{\text{warehouse}}, g^*_1, \cdots, g^*_m\}$$

which obtains a number of $S_{\text{warehouse}}$-objects (that are objects at state $S_{\text{warehouse}}$) as well as a number $\#S_{\langle i \rangle}$ of $S_{\langle i \rangle}$-objects, for each $i$. Notice that all the $S_{\langle i \rangle}$-objects are now active.

From now on, the numbers $\#S_{\langle i \rangle}$ do not change anymore.

Next, for some $j$ nondeterministically chosen, the following transition multiset

$$R_{j,\text{begin}} = \{r^*_{1,j}, \cdots, r^*_{m,j}\}$$

is fired. The $S_{\text{warehouse}}$-objects serve as a warehouse in supplying objects that will be modified into some $S_{\langle i,j \rangle}$-objects. As a result, the transition multiset, for each $i$, newly

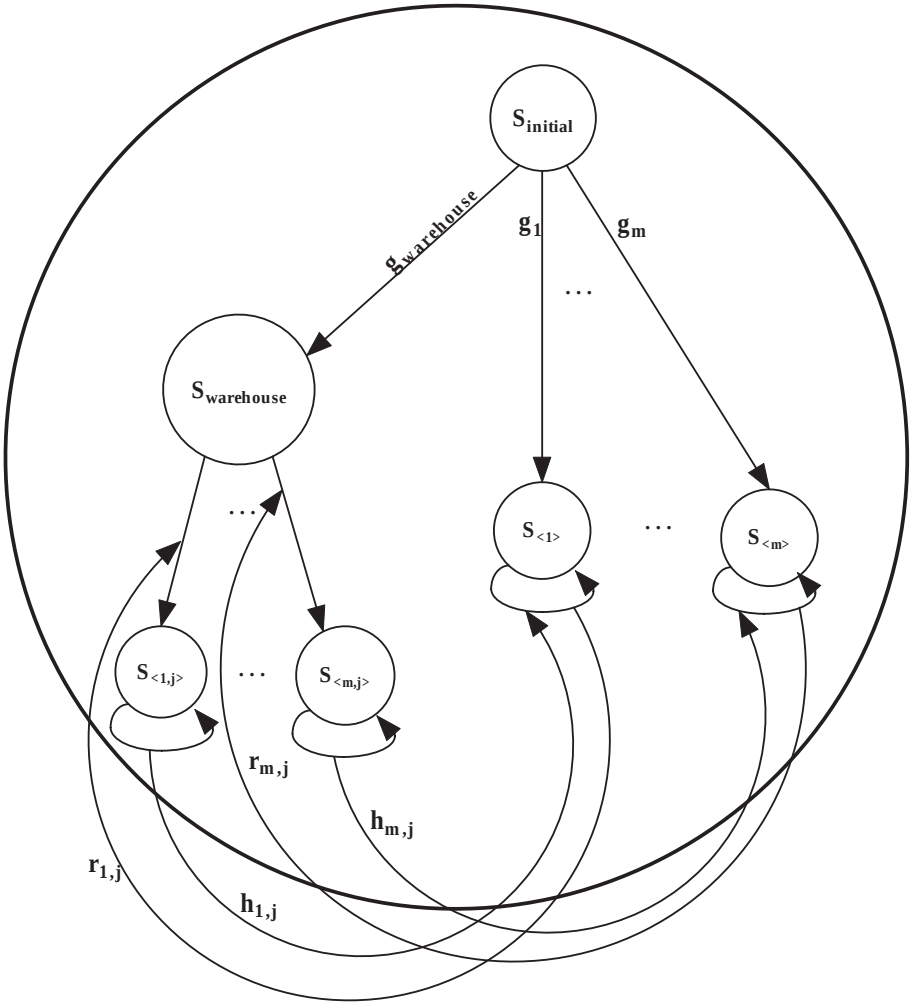


**Fig. 6.** A multiprocess service automaton constructed in the proof of Theorem 5.

adds exactly $\#S_{\langle i,j\rangle}$-objects. (Assuming that the number of $S_{\text{warehouse}}$-objects is not zero; this assumption will be revisited later.) Next, the following transition multiset

$$R_{j,\text{end}} = \{h^*_{1,j}, \cdots, h^*_{m,j}\}$$

will bring all the $S_{\langle i\rangle}$-objects back to being active. Execution of $R_{j,\text{begin}}$ and $R_{j,\text{end}}$ in sequel is called a $j$-round. Clearly, the effect of the $j$-round is: for all $i$, $\#S_{\langle i,j\rangle} := \#S_{\langle i,j\rangle} + \#S_{\langle i\rangle}$. After this round, the automaton $G_{\texttt{mp}}$ goes back to the beginning of this paragraph.

To complete the construction, the set of transition multisets in $G_{\texttt{mp}}$ is

$$\{R_{\text{initial}}, R_{j,\text{begin}}, R_{j,\text{end}} : 1 \leq j \leq m\}.$$

Suppose that

> (*) there exists some run of the automaton $G_{\texttt{mp}}$ from some collection such that the run further satisfies the following property (defined in a moment).

Assume that there are, for $1 \leq j \leq n$, $y_j$ number of $j$-rounds have been performed in the run. It is clear that, currently, $\#S_{\langle i,j\rangle}$ stores $\#S_{\langle i\rangle} \cdot y_j$, for all $i, j$. The aforementioned property is as follows:

> (**) the current values of $\#S_{\langle i,j\rangle}$ and $\#S_{\langle i\rangle}$ satisfy the Presburger formula $\mathcal{P}$ (defined at the beginning of the proof)

conjuncted with $\#S_{\text{warehouse}} > 0$ (recalling the assumption made earlier in this proof that the number of $S_{\text{warehouse}}$-objects is not zero).

One can conclude that the existence in statement (*) is equivalent to the $k$-system in (6) having solutions. The result follows immediately since whether statement (*) holds is simply an instance of the Presburger reachability problem for multiprocess service automata.

□

It is shown in [20] that,

**Theorem 6.** *There is a fixed k such that whether a $k$-system has a solution is undecidable.*

We can directly obtain the following result from Theorem 5 and 6:

**Corollary 1.** *The Presburger reachability problem for multiprocess service automata is undecidable.*

Therefore, from Theorem 1, together with Corollary 1, we can conclude that multiprocess service automata are strictly stronger than (single-process) service automata.

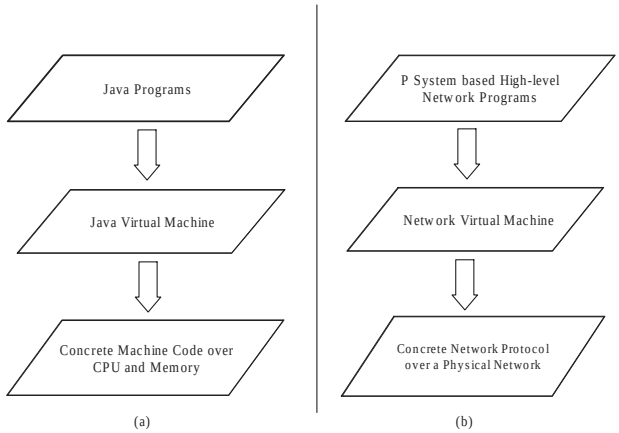


**Fig. 7.** A comparison of Java and P systems based high-level network programs.

## 6 Discussions

Service automata are a form of P systems based high level network programs, running over a network virtual machine. The virtual machine specifies abstract network communicating objects and the operations among the objects. In parallel to the idea of Java Virtual Machine [17], shown in Fig. 7, service automata can be automatically compiled into programs on the network virtual machine and, later, mapped to concrete network protocols on physical networks. Since service automata are independent of the underlying physical networks, similar to Java, they make network applications specified by service automata more portable, easier to verify and test.

The mechanism on the compiler and the mapping is detailedly described in [18]. Specifically, in single-process service automata, for a process running over a network system, there is a *token* passed among the involved objects. There are two ways for an object holding a token to find the next interactive object. One is a completely distributed approach, where the object holding the token finds the next object by broadcasting or multicasting. Another is a hierarchical approach: some servers are maintained to keep the object information. The object holding the token can refer to the servers and know which object it should pass the token to. External transitions are achieved by message exchanges among objects. Except the messages that start the process execution, all other messages in the system are *unicast* messages (i.e., messages are sent from a single source to a single destination). Besides, two objects involved in the same external transition need change the state at the same time, and hence some synchronization is needed. To implement this, we need to ensure that before the two state changes are finished, no other state change can happen. One can achieve this using a some locking mechanism. For multiprocess service automata, one can assign random integers to each objects in the initial active set, and treat those random integers as *Process IDs*. If the sample space of the random numbers is big enough, the probability that two processes have the same process ID is almost 0. To ensure the correctness in a multiprocess service automaton,

it is necessary for us to lock all the objects involved in a process at the beginning and unlock them at the end. In this way, we can treat each process as a single process in a single-process service automaton. However, such an implementation only works for the special case of regular maximal parallelism where the transition multisets $R$ does not contain any $*$. The regular maximal parallelism in general is hard to implement since we are lack of global clocks on a typical asynchronous network.

## References


1. L. Cardelli, G. Ghelli, and A. Gordon. Types for the ambient calculus, 2002.
2. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *IFIP TCS*, pages 333–347, 2000.
3. L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.
4. G. D. Caro and M. Dorigo. Two ant colony algorithms for best-effort routing in datagram networks. In *Proceedings of the Tenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'98)*, pages 541–546, 1998.
5. N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.
6. Z. Dang and O. H. Ibarra. On one-membrane P systems operating in sequential mode. *Int. J. Found. Comput. Sci.*, 16(5):867–881, 2005.
7. M. Dorigo and G. D. Caro. The ant colony optimization meta-heuristic. In *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London, 1999.
8. M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.
9. S. Ginsburg and E. Spanier. Semigroups, presburger formulas, and languages. *Pacific J. of Mathematics.*, 16:285–296, 1966.
10. O. H. Ibarra, H. Yen, and Z. Dang. On various notions of parallelism in P Systems. *Int. J. Found. Comput. Sci.*, 16(4):683-705, 2005.
11. N. Lohmann, P. Massuthe, and K. Wolf. Operating guidelines for finite-state services. In *Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, 2007. Lecture Notes in Computer Science, Vol. 4564, pp. 321-341, Springer, 2007.
12. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
13. Gh. Pãun. Introduction to membrane computing. *See P Systems Web Page at http://psystems.disco.unimib.it*.
14. Gh. Pãun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
15. M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.
16. G. Di Marzo Serugendo, M. Muhugusa, and C. F. Tschudin. A survey of theories for mobile agents. *World Wide Web*, 1(3):139–153, 1998.
17. Sun. Java remote method invocation. *http://java.sun.com/j2se/1.4.2/docs/guide/rmi/*, 2007.
18. Y. Wang. *Clustering, grouping, and process over networks*. PhD Dissertation, Washington State University, 2007.
19. M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.

20. G. Xie, Z. Dang, and O. H. Ibarra. A solvable class of quadratic Diophantine equations with applications to verification of infinite state systems. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, 2003. Lecture Notes in Computer Science, Vol. 2719, pp. 668-680, Springer, 2003.
21. L. Yang, Z. Dang, and O. H. Ibarra. Bond computing systems: a biologically inspired and high-level dynamics model for pervasive computing. In *Proceedings of the 6th International Conference on Unconventional Computation (UC'07)*, 2007. Lecture Notes in Computer Science, Vol. 4618, pp. 226-241, Springer, 2007.