Bond Computing Systems: a Biologically Inspired and High-level Dynamics Model for Pervasive Computing*

Linmin Yang[1], Zhe Dang[1], and Oscar H. Ibarra[2]

[1]School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164, USA
{lyang1,zdang}@eecs.wsu.edu

[2]Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
ibarra@cs.ucsb.edu

Abstract. Targeting at modeling the high-level dynamics of pervasive computing systems, we introduce Bond Computing Systems (BCS) consisting of objects, bonds and rules. Objects are typed but addressless representations of physical or logical (computing and communicating) entities. Bonds are typed multisets of objects. In a BCS, a configuration is specified by a multiset of bonds, called a collection. Rules specify how a collection evolves to a new one. A BCS is a variation of a P system introduced by Gheorghe Paun where, roughly, there is no maximal parallelism but with typed and unbounded number of membranes, and hence, our model is also biologically inspired. In this paper, we focus on regular bond computing systems (RBCS), where bond types are regular, and study their computation power and verification problems. Among other results, we show that the computing power of RBCS lies between LBA (linearly bounded automata) and LBC (a form of bounded multicounter machines) and hence, the regular bond-type reachability problem (given an RBCS, whether there is some initial collection that can reach some collection containing a bond of a given regular type) is undecidable. We also study a restricted model (namely, B-boundedness) of RBCS where the reachability problem becomes decidable.

1 Introduction

As inexpensive computing devices (e.g., sensors, cellular phones, PDAs, etc.) become ubiquitous in our daily life, pervasive computing [14], a proposal of building distributed software systems from (a massive number of) such devices that are pervasively hidden in the environment, is no longer a dream. An example of pervasive computing systems is Smart Home [7], which aims to provide a set of intelligent home appliances that make our life more comfortable and efficient. In Smart Home, a classic scenario is that as soon

* The work by Linmin Yang and Zhe Dang was supported in part by NSF Grant CCF-0430531. The work by Oscar H. Ibarra was supported in part by NSF Grant CCF-0430945 and CCF-0524136.

as a person goes back home from the office, the meal is cooked and the air conditioner is turned on, etc. Another example is a health monitoring network for, e.g., assisting the aged people [1]. In the system, a typical scenario (which is also used throughout this paper) is that when one gets seriously sick, a helicopter will, automatically, come and carry him/her to a clinic.

However, how to design a complex pervasive computing application remains a grand challenge. A way to respond to the challenge is to better understand, both fundamentally and mathematically, some key views of pervasive computing. Following this way, formal computation models reflecting those views are highly desirable and, additionally, can provide theoretical guidelines for further developing formal analysis (such as verification and testing) techniques for pervasive computing systems.

At a high-level, there are at least two views in modeling a pervasive computing system. From the local view, one needs to formally model how a component or an object interacts with the environment with the concept, which is specific to pervasive computing, of context-awareness [12] in mind using notions like Context UNITY [10] and Context-Aware Calculus [15]. From the global view, one needs to formally model how a group of objects evolve and interact with others, while abstracting the lower-level communications away. An example of such a view is the concept of *communities* in the design frame work PICO [13, 4], where objects are (dynamically) grouped and collaborated to perform application-specific services. Inspired by the communities concept in PICO, we introduce a formal computation model in this paper, called *bond computing systems* (BCS), to reflect the global view for pervasive computing.

The basic building blocks of a BCS are *objects*, which are logical representations of physical (computing and communicating) entities. An object is typed but not addressed (i.e., without an individual identifier; we leave addressing to the lower-level (network) implementation). For instance, when a person is seriously sick, we care about *whether* a helicopter will come instead of *which* helicopter will come, under the simplest assumption that all helicopters share the same type *helicopter*.

In a BCS, objects are grouped in a *bond* to describe how objects are associated. For instance, a helicopter carrying a patient can be considered a bond (of two objects). Later, when the helicopter picks up one more patient, the bond then contains three objects. Since objects of the same type share the same name, a bond is essentially a multiset of (typed) objects. A bond itself also has a type to restrict how many and what kinds of objects that can be in the bond. In our system, bonds do not overlap; i.e., one object cannot appear in more than one bond at the same time (we leave overlapping and nested bonds in future work).

In a BCS, a configuration is specified by a multiset of bonds, called a *collection*. The BCS has a number of *rules*, each of which specifies how a collection evolves to a new one. Basically, a rule consists of a number of atomic rules that are fired in parallel, and each atomic rule specifies how objects transfer from a typed bond to another.

Our bond computing systems are a variation of P systems introduced by Gheorghe Paun [8]. A P system is an unconventional computing model motivated from natural phenomena of cell evolutions and chemical reactions. It 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, maximally parallel manner, transitions between the system configurations can be obtained. A sequence of transitions shows how the system is evolving.

Our bond computing systems can be considered as P systems and hence are also biologically inspired. The important features of a BCS, which are tailored for pervasive computing, are the following:

1. The objects in a BCS can not evolve into other objects (e.g., a *helicopter* object can not evolve into a *doctor* object in a BCS). This is because in pervasive computing systems, objects while offering services, mostly maintain their internal structures.
2. The number of bonds in a BCS is unbounded. This makes it flexible to specify a pervasive application with many instances of a community.
3. Each rule in a BCS is global, since the BCS is intended to model the dynamics of a pervasive computing system as a whole rather than a specific bond. If applicable, every two bonds can communicate with each other via a certain rule. In tissue P systems [6, 9], membranes are connected as a directed graph, and one membrane can also communicate with others if there are rules between them, without the constraint of tree-like structures. However, in tissue P systems, every rule can only connect two designated membranes, and in BCS, bonds are addressless and hence one rule can be applied to multiple pairs of bonds. Furthermore, the number of membranes in a tissue P system is bounded.
4. There is no maximal parallelism in a BCS. Maximal parallelism, like in P systems, is extremely powerful [8, 3]. However, considering a pervasive computing model that will eventually be implemented over a network, the cost of implementing the maximal parallelism (which requires a global lock) is unlikely realistic, and is almost impossible in unreliable networks [5]. Hence, rules in a BCS fire asynchronously, while inside a rule, some local parallelism is allowed. Notice that P systems without maximal parallelism have been studied, e.g., in [2], where the number of membranes is fixed, while in BCS, the number of nonempty bonds could be unbounded.

In this paper, we focus on *regular bond computing systems* (RBCS), where bond types are regular, and study their computation power and verification problems. Among other results, we show that the computing power of RBCS lies between LBA (linearly bounded automata) and LBC (a form of bounded multicounter machines) and hence, the following *bond-type reachability problem* is undecidable: whether, for a given RBCS and a regular bond type L, there is some initial collection that can reach some collection containing a bond of type L. Given this undecidability result, we then study a form of restriction that can be respectively applied on RBCS such that the bond-type reachability problem becomes decidable. This form of restricted RBCS is called B-bounded, where each bond contains at most B objects. In this case, the bond-type reachability problem is decidable by showing B-bounded RBCS can be simulated by vector addition systems (i.e., Petri nets).

The rest of the paper is organized as follows. In Section 2, we give the definition of bond computing systems along with an example. In Section 3, we study the computing power of regular bond computing systems and show that the bond-type reachability problem is undecidable in general for regular bond computing systems. In Section 4, we study a restricted form of regular bond computing systems such that the reachability problem becomes decidable. We conclude the paper in Section 5.

2 Definitions

Let $\Sigma = \{a_1, ..., a_k\}$ be an alphabet of symbols, where $k \geq 1$. An instance of a symbol is called an *object*. A *bond* w is a finite multiset over Σ, which is represented by a string in $a_1^*...a_k^*$. For example, the bond $w = a_2^3 a_4^2 a_7^5$ is the multiset consisting of three a_2's, two a_4's, and five a_7's. The empty (null) multiset is denoted by the null string λ.

Throughout this paper, a language L over Σ will mean a subset of $a_1^*...a_k^*$ (representing a set of multisets over Σ). $L_\emptyset$ will denote the language containing only λ. Hence, a *bond type*, that is a (possibly infinite) set of finite multisets, is simply a language L. Let $\mathcal{L} = \{L_\emptyset, L_1, ..., L_l\}$ be $l + 1$ languages (i.e., bond types), where $l \geq k$, and for $1 \leq i \leq k$, $L_i = \{a_i\}$ (i.e., L_i is the language containing only the symbol a_i). We will usually denote a regular language by a regular expression, e.g., a_i instead of $\{a_i\}$.

A *bond collection* $\mathcal{C}$ is an infinite multiset of bonds, where there are only finitely many nonempty bonds. A bond in the collection may belong to one or more bond types in $\mathcal{L}$ or may not belong to any bond type in $\mathcal{L}$ at all.

Example 1. Consider the alphabet $\Sigma = \{p, h, c, d\}$, where p, h, c and d represent *patient, helicopter, clinic* and *doctor*, respectively. We also have bond types $\mathcal{L} = \{L_\emptyset, L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8, L_9, L_{10}\}$, where each L_i is specified by a regular expression as follows:

- By definition, $L_1 = p$, $L_2 = h$, $L_3 = c$ and $L_4 = d$.
- $L_5 = p^*h$ is to specify a "helicopter-patients" bond (i.e., a helicopter carrying 0 or more patients).
- $L_6 = p^{\leq 8}c(d + dd)$ is to specify a "clinic-patients-doctors" bond (i.e., a clinic housing at most 8 patients and with 1 or 2 doctors).
- $L_7 = p^8cd^2$ is to specify a "clinic-patients-doctors" bond that has 8 patients and 2 doctors exactly.
- $L_8 = p^8cd^3$ is to specify a "clinic-patients-doctors" bond that has 8 patients and 3 doctors exactly.
- $L_9 = cd$ is used to specify a "clinic-patients-doctors" bond but with one doctor and no patient.
- $L_{10} = p^{>8}cd^{>2}$ is to specify a "clinic-patients-doctors" bond but with more than 8 patients and more than 2 doctors.

A bond collection $\mathcal{C}$ specifies a scenario of how patients, helicopters and clinics are bonded together. For instance, let $\mathcal{C}$ be a multiset of bonds

$$\{p, p, h, d, d, p^2h, ph, p^4cd, p^8cd^2\}$$

(we omit all the empty bonds λ when writing $\mathcal{C}$). This $\mathcal{C}$ indicates that we have totally 17 patients, 3 helicopters, 2 clinics, and 5 doctors, where one helicopter carries 2 patients, one helicopter carries 1 patient, and the remaining helicopter carries none. Also in the collection, one clinic has 4 patients and 1 doctor (while the other one has 8 patients and two doctors). This collection is depicted in the left hand side of Figure 1. Notice that a bond does not necessarily mean a "physical bond" (like a patient boarding a helicopter). For instance, the patient and the insurance company with which the patient is subscribed are "logically" (instead of "physically") bonded. □

A *rule* specifies how objects are transferred between bonds with certain types. Formally, a *rule* R consists of a finite number of *atomic rules* r, each of which is in the following form:

$$(L)_{\mathrm{left}_r} \xrightarrow{\alpha_r} (L')_{\mathrm{right}_r}, \tag{1}$$

where left_r and right_r are numbers, called the left index and the right index, respectively, L and L' are bond types in $\mathcal{L}$, and string α_r represents the finite multiset of objects that will be transferred.

The semantics of the rule R, when applied on a collection $\mathcal{C}$ of bonds, is as follows. We use I to denote the set of all the left and right indices that appear in the atomic rules r in R. Without loss of generality, let $I = \{1, \cdots, m\}$ for some m.

For each $i \in I$, let w_i be a distinct bond in the collection. We say that the rule R *modifies* $\langle w_1, \cdots, w_m \rangle$ into $\langle w'_1, \cdots, w'_m \rangle$, when all of the following conditions are satisfied:

- For each appearance $(L)_i$, with $L \in \mathcal{L}$ and $i \in I$, in the rule R, w_i must be of bond type L.
- For each $i \in I$, we use LEFT_i to denote the set of all atomic rules r in R such that the left index left_r of r equals i. Then, each w_i must contain all the objects that ought to be transferred, i.e., the multiset union of multisets α_r in each atomic rule r that is in LEFT_i.
- The bonds $\langle w'_1, \cdots, w'_m \rangle$ are the result of $\langle w_1, \cdots, w_m \rangle$ after, *in parallel* for all atomic rules r in R, transferring objects in multiset α_r from bond w_{left_r} to bond w_{right_r}. That is, $\langle w'_1, \cdots, w'_m \rangle$ are the outcome of the following pseudo-code:

  For each $r \in R$
  $w_{\mathrm{left}_r} := w_{\mathrm{left}_r} - \alpha_r$; // '$-$' denotes multiset difference
  For each $r \in R$
  $w_{\mathrm{right}_r} := w_{\mathrm{right}_r} + \alpha_r$; // '$+$' denotes multiset union
  For each $i \in I$
  $w'_i := w_i$.

When there are some $\langle w_1, \cdots, w_m \rangle$ in the collection $\mathcal{C}$ such that R modifies $\langle w_1, \cdots, w_m \rangle$ into some $\langle w'_1, \cdots, w'_m \rangle$, we say that R is *enabled* and, furthermore, R *modifies* the collection $\mathcal{C}$ into collection $\mathcal{C}'$, written

$$\mathcal{C} \xrightarrow{R} \mathcal{C}'$$

where $\mathcal{C}'$ is the result of replacing $w_1, \cdots, w_m$ in $\mathcal{C}$ with $w'_1, \cdots, w'_m$, respectively.

Example 2. Let bond collection $\mathcal{C}$ be the one defined in Example 1. That is, $\mathcal{C}$ is the multiset of bonds $\{p, p, h, d, d, p^2h, ph, p^4cd, p^8cd^2\}$. Recall that p, h, c and d represent patient, helicopter, clinic and doctor, respectively. Consider a rule, called TRANSFER_CALL-IN, that describes the scenario that a helicopter transfers a patient to a clinic that already houses 8 patients but with only two doctors on duty. In parallel with the transferring, an on-call doctor (who is not bonded with a clinic yet) is called in and bonded with the clinic to treat the patient. The rule TRANSFER_CALL-IN consists of the following two atomic rules:

TRANSFER: $(p^*h)_1 \xrightarrow{p} (p^8cd^2)_2$
CALL-IN: $(d)_3 \xrightarrow{d} (p^8cd^2)_2$

In order to describe how the rule is fired, by definition, we (nondeterministically) choose three bonds from the collection $\mathcal{C}$: $\langle w_1, w_2, w_3\rangle = \langle p^2h, p^8cd^2, d\rangle$. Notice that bond w_1 corresponds to the term $(p^*h)_1$ in the atomic rule TRANSFER; bond w_2 corresponds to the term $(p^8cd^2)_2$ in the atomic rule TRANSFER and in the atomic rule CALL-IN; and bond w_3 corresponds to the term $(d)_3$ in the atomic rule CALL-IN. (Note that the subscript i of w_i equals the subscript of the term correspondent to the bond w_i.) One can easily check that these three bonds are respectively of the bond types specified in the terms. Firing the rule on the chosen $\langle w_1, w_2, w_3\rangle$ will make a p transferred from bond w_1 to w_2 and make a d transferred from bond w_3 to w_2, in parallel. That is, $\langle w_1, w_2, w_3\rangle$ is modified into $\langle w'_1, w'_2, w'_3\rangle = \langle ph, p^9cd^3, \lambda\rangle$. Replacing the previously chosen $\langle w_1, w_2, w_3\rangle$ in $\mathcal{C}$ with the $\langle w'_1, w'_2, w'_3\rangle$, we get $\mathcal{C}' = \{p, p, h, d, ph, ph, p^4cd, p^9cd^3\}$ (again, we omit the empty bonds λ). Hence, we have $\mathcal{C} \xrightarrow{R} \mathcal{C}'$ (where R is the rule TRANSFER_CALL-IN) as depicted in Figure 1. Notice also that, depending on the choice of $\langle w_1, w_2, w_3\rangle$, the resulting collection $\mathcal{C}'$ may not be unique; e.g., one can choose $\langle w_1, w_2, w_3\rangle = \langle ph, p^8cd^2, d\rangle$ from $\mathcal{C}$ and, subsequently, $\mathcal{C}' = \{p, p, h, d, p^2h, h, p^4cd, p^9cd^3\}$. Finally, we shall point out that if we change the rule TRANSFER_CALL-IN into the following form:

TRANSFER: $(p^*h)_1 \xrightarrow{ppp} (p^8cd^2)_2$
CALL-IN: $(d)_3 \xrightarrow{d} (p^8cd^2)_2$

then this rule is not enabled under the configuration $\mathcal{C}$. (Since, now, the rule, when fired, *must* transfer 3 patients from a bond of type p^*h but we do not have such type of bond in $\mathcal{C}$ which contains at least 3 patients to transfer) □

A bond type L is *regular* if L is a regular language (i.e., a regular language in $a_1^*...a_k^*$). In particular, when L is (regular expression) a, we say that L is the a-free bond type. The object (a) in the a-free bond is called a *free* object.

A *bond computing system (BCS)* M is a machine specified by the following tuple:

$$\langle \Sigma, \mathcal{L}, \mathcal{R}, \theta\rangle, \tag{2}$$

where $\mathcal{L}$ is a finite set of bond types, $\mathcal{R}$ is a finite set of rules, and θ is the *initial constraint*. An *initial* collection of M is one in which a nonempty bond must be of a-free bond type for some a. That is, every object is a free object in an initial collection.

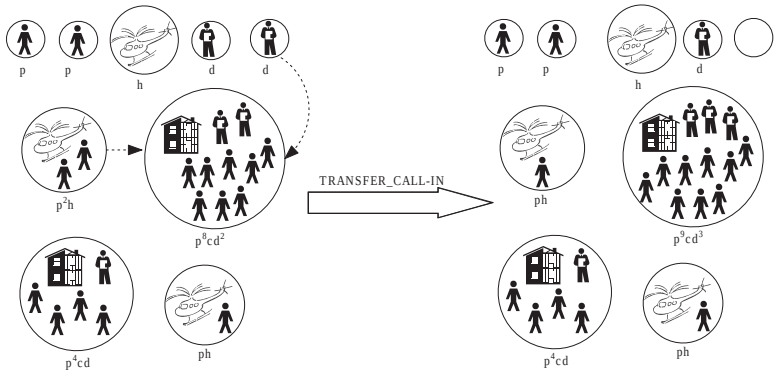

Fig. 1. Collection $\mathcal{C}$ (the left hand side) is modified into collection $\mathcal{C}'$ (the right hand side) after firing the rule TRANSFER_CALL-IN described in Example 2. The dotted arrows in the left hand side of this figure indicate the bonds on which the two atomic rules TRANSFER and CALL-IN (in the rule TRANSFER_CALL-IN) fire upon in parallel.

Additionally, the initial collection must satisfy the initial constraint θ which is specified by a mapping $\theta : \Sigma \to \mathrm{N} \cup \{*\}$. That is, for each $a \in \Sigma$ with $\theta(a) \neq *$, the initial collection must contain exactly $\theta(a)$ number of free a-objects (while for a with $\theta(a) = *$, the number of free a-objects in the initial collection could be any number). Hence, the initial constraint θ is to specify the symbols in Σ whose initial multiplicities are fixed. For two collections $\mathcal{C}$ and $\mathcal{C}'$, we say that $\mathcal{C}$ can *reach* $\mathcal{C}'$, written

$$\mathcal{C} \rightsquigarrow_M \mathcal{C}'$$

if, for some n,

$$\mathcal{C}_0 \overset{R_1}{\Longrightarrow} \mathcal{C}_1 \cdots \overset{R_n}{\Longrightarrow} \mathcal{C}_n$$

where $\mathcal{C} = \mathcal{C}_0, \mathcal{C}_1, \cdots, \mathcal{C}_n = \mathcal{C}'$ are collections, and $R_1, \cdots, R_n$ are rules in $\mathcal{R}$.

In this paper, we focus on *regular* bond computing system (RBCS) M where each $L \in \mathcal{L}$ is regular. As we mentioned earlier, a BCS M can be thought of a design model of a pervasive software system. An important issue in developing such a design, in particular for safety-critical and/or mission critical systems, is to make sure that the software system does not have some undesired behaviors. This issue is called *(automatic) verification*, which seeks an (automatic) way to verify that a system satisfies a given property.

Example 3. Let M be a regular bond computing system with $\Sigma = \{p, h, c, d\}$ and $\mathcal{L}$ specified in Example 1. The initial constraint is defined as follows: $\theta(p) = *$, $\theta(h) = 3$, $\theta(c) = 2$, and $\theta(d) = 5$. The rules in $\mathcal{R}$ are in below:

- R_1: $(p)_1 \overset{p}{\to} (p^*h)_2$;
- R_2 consisting of two atomic rules:
  r_1: $(p^*h)_1 \overset{p}{\to} (p^8cd^2)_2$;
  r_2: $(d)_3 \overset{d}{\to} (p^8cd^2)_2$;

- R_3: $(p^*h)_1 \xrightarrow{p} (p^{\leq 8}c(d+dd))_2$;
- R_4: $(p^*h)_1 \xrightarrow{p} (p^{>8}cd^{>2})_2$;
- R_5 consisting of two atomic rules:
  r_1: $(p^8cd^3)_1 \xrightarrow{p} (\lambda)_2$;
  r_2: $(p^8cd^3)_1 \xrightarrow{d} (\lambda)_3$;
- R_6: $(p^{\leq 8}c(d+dd))_1 \xrightarrow{p} (\lambda)_2$;
- R_7: $(p^{>8}cd^{>2})_1 \xrightarrow{p} (\lambda)_2$;
- R_8: $(d)_1 \xrightarrow{d} (c)_2$;
- R_9: $(d)_1 \xrightarrow{d} (cd)_2$.

In above, R_1 indicates that a patient boards a helicopter. R_2, R_3 and R_4 indicate that a patient is transferred from a helicopter to a clinic, where, in particular, R_2 means that the clinic does not have enough doctors, so an additional doctor is called in. R_5, R_6 and R_7 indicate that a patient leaves the clinic (after being cured). In particular, R_5 means that, when the clinic does not have many patients, a doctor can be off-duty. R_8 and R_9 are to fill a clinic (without patient) with one or two doctors.

□

A simplest (and, probably, the most important) class of verification queries concern *reachability*, which has various forms. A *collection-to-collection reachability* problem is to decide whether, given two collections, one can reach the other in a given BCS:

Given: an RBCS M and two collections $\mathcal{C}$ and $\mathcal{C}'$ (where $\mathcal{C}$ is initial),

Question: $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$?

Observe that, since the total number of objects in M does not change in any run of M from the initial collection $\mathcal{C}$, the collection-to-collection reachability problem is clearly decidable in time polynomial in the size of M and the size of $\mathcal{C}$ (the total number of objects in $\mathcal{C}$, where the number is *unary*). Just use "breadth-first" to find all reachable collections.

Theorem 1. *The collection-to-collection reachability problem for regular bond computing systems is decidable in polynomial time.*

One can also generalize Theorem 1 to the *generalized collection-to-collection reachability* problem, where the initial collection is still given but the target collection is not:

Given: an RBCS M, a regular bound type L, and an initial collection $\mathcal{C}$,

Question: is there a collection $\mathcal{C}'$ such that $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$ and $\mathcal{C}'$ contains a bond of type L ?

Theorem 2. *The generalized collection-to-collection reachability problem for regular bond computing systems is decidable in polynomial time.*

Remark 1. It is easily verified that the two theorems above hold even when the bond types are not regular, but are in PRIME; i.e., the class of languages accepted by deterministic Turing machines in polynomial time.

In the collection-to-collection reachability problems studied in Theorems 1 and 2, M starts with a given initial collection. Therefore, the decision algorithms for the problems essentially concern testing: each time, the decision algorithms only verify M under a concrete initial collection. It is desirable to verify that the system M has the undesired behavior for *some* initial collections. That is, considering the negation of the problem, we would like to know whether the design is correct under every possible (implementation) instance.

To this end, we will study the following *bond-type reachability* problem, where the initial collection is not given:

Given: an RBCS M and a regular bound type L,

Question: are there two collections $\mathcal{C}$ and $\mathcal{C}'$ such that $\mathcal{C}$ is initial, $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$, and $\mathcal{C}'$ contains a bond of type L ?

The answer of the above question is meaningful in many aspects. One is that it can help decide whether a system is well designed. Suppose the bond type L is something undesired which we do not want to have in a system. If we have $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$, and $\mathcal{C}'$ contains a nonempty bond of type L, we say that the system is not well designed.

Example 4. Consider the bond computing system M in example 3 and with $\mathcal{C} = \{p, p, h, d, d, p^2h, ph, p^4cd, p^8cd^2\}$ reached at some moment. Suppose that $L = p^{>8}cd^{\leq 2}$ is an "undesired" bond type (e.g., a clinic with too many patients but with too few doctors). First, we choose $\langle w_1, w_2\rangle = \langle p, p^2h\rangle$ from $\mathcal{C}$. After applying R_1, $\langle w_1', w_2'\rangle = \langle \lambda, p^3h\rangle$ and we get $\mathcal{C}' = \{p, h, d, d, p^3h, ph, p^4cd, p^8cd^2\}$, or, $\mathcal{C} \xrightarrow{R_1} \mathcal{C}'$. Next, we pick $\langle w_1, w_2\rangle = \langle p^3h, p^8cd^2\rangle$ from $\mathcal{C}'$, after applying R_3, $\langle w_1', w_2'\rangle = \langle p^2h, p^9cd^2\rangle$, we get $\mathcal{C}'' = \{p, h, d, d, p^2h, ph, p^4cd, p^9cd^2\}$, or, $\mathcal{C}' \xrightarrow{R_3} \mathcal{C}''$. Therefore, $\mathcal{C} \rightsquigarrow_M \mathcal{C}''$, and in $\mathcal{C}''$ there is a bond p^9cd^2, which is a bond of the undesired bond type L. Therefore, M is not designed as expected. □

In the following sections, we are going to investigate the computing power of regular bond computing systems. Our results show that the bond-type reachability problem is undecidable in general. However, there is an interesting special case where the problem becomes decidable.

3 Undecidability of Bond-type Reachability

To show that the bond-type reachability problem is undecidable, we first establish that the computing power of RBCS (regular bond computing systems) lies between LBA (linearly bounded automata) and LBC (linearly bounded multicounter machines). To proceed further, some definitions are needed.

An LBA A is a Turing machine where the R/W head never moves beyond the original input region. For the purpose of this paper, we require that the LBA works on bounded input; i.e., input in the form of $b_1^* \cdots b_m^*$. As usual, we use $\mathrm{Lang}(A)$ to denote the set of input words that are accepted by A.

Let M be an RBCS. Recall that, in M, the initial constraint θ is used to specify the case when some symbols in Σ whose initial multiplicities are fixed (and specified by θ). Without loss of generality, we use $a_1, \cdots, a_l$ (for some $l \leq k$) to denote the

remaining symbols and, by ignoring those "fixed symbols", we simply use $a_1^{i_1} \cdots a_l^{i_l}$ to denote an initial collection $\mathcal{C}$ of M where each object is a free object. Let L be a given regular bond type. We say that the initial collection is *L-accepted* by M if there is some collection $\mathcal{C}'$ that contains a bond of type L such that $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$. As usual, we use $\mathrm{Lang}(M, L)$ to denote the set of all words $a_1^{i_1} \cdots a_l^{i_l}$ that are L-accepted by M. Notice that the bond type reachability problem is equivalent to the emptiness problem for $\mathrm{Lang}(M, L)$.

We first observe that, since the total "size" (i.e., the total number of objects in the system) of a collection always equals the size of the initial collection during a run of M, an RBCS M can be simulated by an LBA.

Theorem 3. *For any given regular bond computing system M and a regular bond type L, one can construct a linearly bounded automaton A such that* $\mathrm{Lang}(M, L) = \mathrm{Lang}(A)$.

Remark 2. Clearly, the theorem above holds, even when the bond types are not regular, but are context-sensitive languages (= languages accepted by LBAs).

Remark 3. Actually, LBAs are strictly stronger than RBCS (since languages accepted by RBCS are upper-closed (i.e., if $a_1^{i_1} \cdots a_k^{i_k}$ is accepted then so is $a_1^{i'_1} \cdots a_k^{i'_k}$, for any $i_j \leq i'_j$) and this is not true for LBAs).

A multicounter machine is a (nondeterministic) program P equipped with a number of nonnegative integer counters $X_1, \cdots, X_n$ (for some n). Each counter can be incremented by one, decremented by one, and tested for 0, along with state transitions. That is, P consists of a finite number of instructions, each of which is in one of the following three forms:

$s : X++, \texttt{goto}\ s'$;
$s : X--, \texttt{goto}\ s'$;
$s : X == 0?, \texttt{goto}\ s'$;

where X is a counter. Initially, the counters start with 0. During a run of the machine P, it crashes when a counter falls below 0. Notice that, in our instruction set, there is no instruction in the form of $s : X > 0?, \texttt{goto}\ s'$. In fact, this instruction can be simulated by the following two instructions: $s : X--, \texttt{goto}\ s''$ and $s'' : X++, \texttt{goto}\ s'$, where s'' is a new state. The machine halts when it enters a designated accepting state. It is well known that multicounter machines are Turing-complete (even when there are two counters). We say that a nonnegative integer tuple $(x_1, \cdots, x_n)$ is *accepted* by P if P has an accepting run during which the maximal value of counter X_i is bounded by x_i for each i. One can treat the tuple $(x_1, \cdots, x_n)$ as an (input) word (where each number is unary). In this way, we use language $\mathrm{Lang}(P)$ to denote the set of all nonnegative integer tuples accepted by P. Notice that, in here, counter values in the multicounter machine can not exceed the values given in the input. Therefore, we call such P as a *linearly bounded multicounter machine* (LBC).

Our next result shows that an LBC P can be simulated by an RBCS M. A rule that contains only one atomic rule is called *simple*. We call M *simple* if every rule in M is simple. In a collection $\mathcal{C}$, a bond is *trivial* if it is either empty or of a free bond type; that is, the bond contains at most one (free) object. When there are at most m

nontrivial bonds in any reachable collection during any run of M (starting from any initial collection), we simply say that M *contains* m nontrivial bonds. In fact, the result holds even when M is simple and contains 2 nontrivial bonds.

Theorem 4. *For any given linearly bounded multicounter machine P, one can construct an RBCS M, which contains at most 2 nontrivial bonds, and a regular bond type L such that* $\mathrm{Lang}(P) = \mathrm{Lang}(M, L)$.

Proof. We only prove the result for $n = 2$ (i.e., P has two counters). The proof can be easily generalized for more counters. Let P be a linearly bounded multicounter machine with two counters X and Y and with states $s_0, \cdots, s_m$ (assuming that s_0 being the initial state and s_m being the accepting state). Without loss of generality, we assume that, for each (ordered) state pair $\langle s_i, s_j \rangle$, there is at most one instruction that leads from s_i to s_j in P. We now construct the desired RBCS M which has the following symbols:

- f_{ij}, $1 \leq i, j \leq m$, are symbols for state pairs $\langle s_i, s_j \rangle$ in two counter machine P;
- a and b are the symbols for counter X and counter Y in P (will be explained in a moment), respectively;
- c is the symbol for states (will be explained in a moment);
- d, e and g are auxiliary symbols.

There are only two nontrivial bonds in P. One is called the eg-bond, which must contain eg and is in the form of $ega^*b^*c^*$ or in the form of $ega^*b^*c^*f_{ij}$, for some i, j. The other is called the d-bond, which must contain d and is in the form of

$$da^*b^*c^*f_{00} \cdots f_{ij} \cdots f_{mm}$$

or in the same form but with one state pair symbol f_{ij} dropped.

Intuitively, counter X (resp. Y) in two-counter machine P is encoded as the multiplicity of symbol a (resp. b) in the d-bond, while the current state s_i of the two-counter machine is encoded as the multiplicity of symbol c in the d-bond. On the other hand, the eg-bond serves as the "warehouse" of objects that will be supplied and stored when a counter instruction is run in P.

We now show how to construct simple rules in M with respect to a counter instruction in P (Without loss of generality, we only show the instructions for counter X.):

(A). For counter instruction

$$s_i : X++, \texttt{goto}\ s_j$$

in P, we have the following two cases to consider

- When $i \leq j$, we include the following simple rule in M (for simplicity, we drop the left and right indices in the rule since there is only one atomic rule in a simple rule):

  $$(ega^*b^*c^*) \overset{ac^{j-i}}{\rightarrow} (da^*b^*c^i f_{00} \cdots f_{ij} \cdots f_{mm}).$$

  That is, when the current state is s_i (encoded as the c^i in the d-bond in above), one a and $(j - i)$ copies of c flow from the eg-bond to the d-bond (to simulate $X++$ and $s_i \rightarrow s_j$ state transition).

– When $i > j$, we include two simple rules in M. The first simple rule is

$$(da^*b^*c^i f_{00} \cdots f_{ij} \cdots f_{mm}) \stackrel{c^{i-j} f_{ij}}{\rightarrow} (ega^*b^*c^*),$$

and the second simple rule is

$$(ega^*b^*c^* f_{ij}) \stackrel{af_{ij}}{\rightarrow} (da^*b^*c^j f_{00} \cdots f_{i(j-1)} f_{i(j+1)} \cdots f_{mm}).$$

The first simple rule implements the state transition ($s_i \rightarrow s_j$) by moving $(i-j)$ copies of c from the d-bond to the eg-bond. The second simple rule implements the counter increment ($X++$) by moving one a from the eg-bond back to the d-bond. Notice that, in the bond computing system M, there is no state. That is, executions of rules are not ordered in advance. Therefore, in the construction, we employ f_{ij} as a "messenger" that forces these two simple rules run in the expected order (and will not interfere with other rules as well).

(B). For counter instruction

$$s_i : X--, \texttt{goto}\ s_j,$$

similar to the $X++$ instruction in above, we also have the following two cases to consider

– When $i \geq j$, we include the following simple rule in M:

$$(da^*b^*c^i f_{00} \cdots f_{ij} \cdots f_{mm}) \stackrel{ac^{i-j}}{\rightarrow} (ega^*b^*c^*).$$

– When $i < j$, we include the following two simple rules in M:

$$(da^*b^*c^i f_{00} \cdots f_{ij} \cdots f_{mm}) \stackrel{af_{ij}}{\rightarrow} (ega^*b^*c^*)$$

and

$$(ega^*b^*c^* f_{ij}) \stackrel{c^{j-i} f_{ij}}{\rightarrow} (da^*b^*c^i f_{00} \cdots f_{i(j-1)} f_{i(j+1)} \cdots f_{mm}).$$

(C). For counter instruction

$$s_i : X == 0?, \texttt{goto}\ s_j,$$

we have two cases to consider:

– When $i \geq j$, we include the following simple rule in M:

$$(db^*c^i f_{00} \cdots f_{ij} \cdots f_{mm}) \stackrel{c^{i-j}}{\rightarrow} (ega^*b^*c^*).$$

Notice that there is no a (i.e., $X == 0$) in the LHS of the rule.

– When $i < j$, we include the following simple rule:

$$(ega^*b^*c^*) \stackrel{c^{j-i}}{\rightarrow} (db^*c^i f_{00} \cdots f_{ij} \cdots f_{mm}).$$

Similarly, notice that there is no a (i.e., $X == 0$) in the RHS of the rule.

The initial constraint of M is that there are m copies of c-objects, one d-object, one e-object, and one g-object. Since an initial collection of M only contains free objects, we need rules to create the first eg-bond and the first d-bond from which the simulation of the counter machine really starts (using the simple rules created earlier).

For the eg-bond, we have the following simple rules:

$(c) \xrightarrow{c} (e)$;
$(c) \xrightarrow{c} (ec)$;
$\vdots$
$(c) \xrightarrow{c} (ec^{m-1})$;
$(a) \xrightarrow{a} (ea^*b^*c^m)$;
$(b) \xrightarrow{b} (ea^*b^*c^m)$;
$(g) \xrightarrow{g} (ea^*b^*c^m)$.

The first m rules create the bond ec^m. The following two rules populates the bond with a number of a and b objects. The last rule stops the population with the eg-bond created.

For the d-bond, we have the following simple rules:

$(f_{00}) \xrightarrow{f_{00}} (d)$;
$\vdots$
$(f_{0m}) \xrightarrow{f_{0m}} (df_{00} \cdots f_{0(m-1)})$;
$\vdots$
$(f_{mm}) \xrightarrow{f_{mm}} (df_{00} \cdots f_{m(m-1)})$.

In short, these rules create the d-bond $df_{00} \cdots f_{mm}$ (i.e., the initial configuration of the counter machine P with counter values 0 and initial state s_0).

Take bond type L to be $da^*b^*c^m f_{00} \cdots f_{mm}$ which corresponds to the accepting configurations for counter machine P. From the above simulation, it is clear that, for any numbers x and y, the following two items are equivalent:

- P has an accepting on which counters X and Y are bounded by x and y, respectively.
- M, starting from an initial collection with x copies of a and y copies of b, can reach a collection containing a bond of type L.

The result follows. □

An atomic rule in (1) is *single* if there is exactly one object that will be transferred; i.e., the size of α_r is 1. A rule that contains only single atomic rules is also called *single*. An RBCS M is *single* if it contains only single rules. Notice that every simple rule can be rewritten into a single rule. For instance, a simple rule like $(a^+b^*)_1 \xrightarrow{abb} (a^*c^+)_2$ is equivalent to the single rule consisting of the following three atomic single rules:

$(a^+b^*)_1 \xrightarrow{a} (a^*c^+)_2$;
$(a^+b^*)_1 \xrightarrow{b} (a^*c^+)_2$;
$(a^+b^*)_1 \xrightarrow{b} (a^*c^+)_2$.

Therefore, a simple RBCS can be simulated by a single RBCS. Following the same idea, one can also observe that an RBCS can be simulated by a single RBCS. Hence,

combining Theorems 3 (and Remark 3) and Theorem 4, we have the following diagram summarizing the computing power, in terms of language acceptance, of regular bond computing systems (from higher to lower):

$$\text{LBA} \ > \ \text{RBCS} \ = \ \text{single RBCS} \ \geq \ \text{simple RBCS} \ \geq \ \text{LBC}.$$

We currently do not know whether some of the $\geq$'s are strict.

Notice that the emptiness problem for LBC (which is equivalent to the halting problem for multicounter machines) is known undecidable. According to Theorem 4, for (single) RBCS, the emptiness problem (i.e., $\text{Lang}(M, L) = \emptyset$? for a given RBCS M and regular bond type L) is undecidable as well. Hence,

Theorem 5. *The bond-type reachability problem for regular bond computing systems is undecidable. The undecidability remains even when the systems are simple.*

We now generalize the notion of L-acceptance for an RBCS M. Let $L_1, \cdots, L_n$, $Q_1, \cdots, Q_m$ (for some n and m) be bond types.

Similar to L-acceptance defined earlier, we say that an initial collection $\mathcal{C}$ is $(L_1, \cdots, L_n;\ Q_1, \cdots, Q_m)$-*accepted* by M if there is some collection $\mathcal{C}'$ with $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$ such that $\mathcal{C}'$ contains a bond of type L_i for each i and does not contain a bond of type Q_j for each j. As in the definition of $\text{Lang}(M, L)$, we use $\text{Lang}(M, L_1, \cdots, L_n; Q_1, \cdots, Q_m)$ to denote the set of all words $a_1^{i_1} \cdots a_l^{i_l}$ that are $(L_1, \cdots, L_n; Q_1, \cdots, Q_m)$-accepted by M.

Now, Theorem 4 can be generalized to show that an RBCS can simulate a *multihead two-way nondeterministic finite automaton* (multihead 2NFA).

A multihead 2NFA M is a nondeterministic finite automaton with k two-way read-only input heads (for some k). An input string $x \in a_1^*...a_n^*$ (the a_i's are distinct symbols) is accepted by M if, when given $\$x\$$ (the \$'s are the left and right end markers) with M in the initial state and all k heads on the left end marker, M eventually halts in an accepting state with all heads on the right end marker. The language accepted by M is denoted by Lang(M).

Now let P be an LBC with a set C of k counters. Let $n \geq 1$. Partition C into n sets $C_1, ..., C_n$ of counters, where each C_i has k_i counters, and $k = k_1 + ... + k_n$. We say that a nonnegative integer tuple $(x_1, ..., x_n)$ is accepted by P if P has an accepting run during which each of the k_i counters in C_i is bounded by (and achieve) the value x_i. Define Lang(P) to be the set of all n-tuples $(x_1, ..., x_n)$ accepted by P. Note that the definition given earlier can be considered as the current definition with $k_1 = ... = k_n = 1$ but without requiring the bound x_i being achieved.

Slightly modifying the proof of Theorem 4 (using $Q_1, \cdots, Q_m$ to specify that free bond types all disappear at the end of computation), we have

Theorem 6. *For any given linearly bounded multicounter machine P, one can construct an RBCS M, which contains at most 2 nontrivial bonds, and regular bond types $L_1, \cdots, L_n, Q_1, \cdots, Q_m$ such that* $\text{Lang}(P) = \text{Lang}(M, L_1, \cdots, L_n, Q_1, \cdots, Q_m)$.

Actually, LBC and multihead 2NFA are equivalent in the following sense:

Lemma 1. *A set $O \subseteq N^n$ is accepted by an LBC P if and only if the language $\{a_1^{x_1} a_2^{x_2} ... a_n^{x_n} \mid (x_1, ..., x_k) \in O\}$ is accepted by a multihead 2NFA.*

Proof. The "only if" part is obvious. For the "if" part, suppose M is a multihead 2NFA with k heads: $h_1, ..., h_k$. We first construct an intermediate machine – a multitape/multihead 2NFA M' with n inputs (with end markers) and k two-way read-only heads per tape. When M has input $\$a_1^{i_1}...a_n^{i_n}\$$, the n-tapes $T_1, ..., T_k$ of M' have $\$a_1^{i_1}\$, ..., \$a_n^{i_n}\$$, respectively. Each tape T_j ($1 \leq j \leq n$) has heads $h_{1j}, ..., h_{kj}$. We construct M' so that M on input $\$a_1^{i_1}...a_n^{i_n}\$$ accepts if and only if M' with n-tuple $(\$a_1^{i_1}\$, ..., \$a_n^{i_n}\$)$ accepts. Initially every head h_{ij} is on the left end marker of its associated tape. Head h_i of M is simulated using heads $h_{i1}, h_{i2}, ..., h_{in}$ in tapes $T_1, ..., T_n$, respectively. At the start of the computation of M, heads $h_1, ..., h_k$ are on the left end marker. The simulation of M by M' is fairly straightforward. For example, when head h_i of M is operating on the a_j-segment of the input, M' uses head h_{ij} on tape T_j. When h_i moves from the a_j-segment to the a_{j+1}-segment, M' uses head $h_{i(j+1)}$ on tape T_{j+1}. When h_i moves back from the a_{j+1}-segment to the a_j-segment, M' uses again head h_{ij} on tape T_j, etc. From M' we can now easily construct an LBC P with n sets of counters $C_1, ..., C_n$. Each C_j will have $2k$ counters to simulate the k heads on tape T_j of M'. At the start of the computation, k of these counters are incremented to some nondeterministic value i_j. Clearly, two counters can simulate the movement of head h_{ij} on tape T_j containing $\$a_j^{i_j}\$$, where initially, one counter has value i_j and the other counter has value zero. (One counter is decremented while the other is increment. Thus the sum of the values of counters in the pair is always i_j.) We omit the details. □

From Theorem 6, we have the following corollary:

Corollary 1. *For any given multihead 2NFA M, one can construct an RBCS M', which contains at most 2 nontrivial bonds, and regular bond types $L_1, \cdots, L_n, Q_1, \cdots, Q_m$ such that* $\mathrm{Lang}(M) = \mathrm{Lang}(M', L_1, \cdots, L_n, Q_1, \cdots, Q_m)$.

Since multihead 2NFAs are equivalent to $\log n$ space-bounded nondeterministic Turing machines (NTMs), we also obtain the following result:

Corollary 2. *For any given $\log n$ space-bounded NTM M, one can construct an RBCS M', which contains at most 2 nontrivial bonds, and regular bond types $L_1, \cdots, L_n$, $Q_1, \cdots, Q_m$ such that* $\mathrm{Lang}(M) = \mathrm{Lang}(M', L_1, \cdots, L_n, Q_1, \cdots, Q_m)$.

Next, we show that an RBCS can simulate the computation of a nondeterministic linear-bounded automaton (LBA). Let M be an LBA with input alphabet $\Sigma = \{1, 2, ..., k\}$. We represent a string $x = d_n...d_0$ (each d_i in $\{1, ..., k\}$) as an integer $N(x) = d_n k^n + d_{n-1}k^{n-1} + ... + d_1 k^1 + d_0 k^0$. Let o be a symbol. For a language $L \subseteq \Sigma^*$, define Unary(L) = $\{o^{N(x)} \mid x \in L\}$. In [11], it was shown that given an LBA M, one can construct a multihead 2NFA M' such Lang(M') = Unary(Lang(M)). Hence, we have:

Corollary 3. *For any given LBA M, one can construct an RBCS M', which contains at most 2 nontrivial bonds, and regular bond types $L_1, \cdots, L_n, Q_1, \cdots, Q_m$ such that* $\mathrm{Lang}(M) = \mathrm{Lang}(M', L_1, \cdots, L_n, Q_1, \cdots, Q_m)$.

4 Restricted Bond Computing Systems

According to Theorem 5, the bond-type reachability problem is undecidable for regular bond computing systems. In this section, we are going to investigate further restrictions that can be applied such that the problem becomes decidable.

An n-dimensional *vector addition system with states* (VASS) G is a 5-tuple

$$\langle V, p_0, p_f, S, \delta \rangle$$

where V is a finite set of *addition vectors* in 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 δ 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 $u \in \mathrm{N}^n$. 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 N^n, we say that x can *reach* y, written $x \rightsquigarrow_G y$, 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.

Let B be a constant and M be an RBCS. We say that an RBCS M is B-bounded if, in any reachable collection of M (starting from any initial collection), there is no bond containing more than B objects. Essentially, we restrict the size of each bond in M while we do not restrict the number of the nonempty bonds. Notice that there are at most t (depending only on B and k – the size of the alphabet Σ) distinct words (i.e., bonds)

$$w_0, w_1, \cdots, w_t,$$

where each w_i is with size at most B and w_0 is the empty word. Accordingly, we use L_{w_i} to denote the bond type $\{w_i\}$. For a collection $\mathcal{C}$ of the B-bounded M, we use $\#(w_i)$ to denote the multiplicity of type L_{w_i} (nonempty) bonds in $\mathcal{C}$, with $i \geq 1$. In this way, the $\mathcal{C}$ can then be uniquely represented by a vector $\#(\mathcal{C})$ defined as

$$(\#(w_1), \cdots, \#(w_t)) \in \mathrm{N}^t.$$

Accordingly, the reachability relation $\mathcal{C} \rightsquigarrow_M \mathcal{C}'$ for the B-bounded M can then be represented by $\#(\mathcal{C}) \rightsquigarrow_M \#(\mathcal{C}')$; i.e., $\rightsquigarrow_M \subseteq \mathrm{N}^t \times \mathrm{N}^t$.

Our definition of B-boundedness for RBCS M is effective. That is, for any B-bounded RBCS M, one can construct an equivalent B-bounded RBCS M' such that the nonempty bond types in M' are $L_{w_1}, \cdots, L_{w_t}$ and $\rightsquigarrow_M = \rightsquigarrow_{M'}$. The proof is straightforward. So, without loss of generality, we further assume that the bond types in a B-bounded RBCS M are given as $L_{w_1}, \cdots, L_{w_t}$.

Theorem 7. *B-bounded regular bond computing systems can be simulated by vector addition systems with states. That is, for a B-bounded RBCS M, one can construct a VASS G such that $\rightsquigarrow_M = \rightsquigarrow_G$.*

Proof. We need only show that every rule R in M can be faithfully translated to two transitions in G. Then, the result follows.

Consider a rule R consisting of atomic rules r in the form of $(L)_{\mathrm{left}_r} \stackrel{\alpha_r}{\rightarrow} (L')_{\mathrm{right}_r}$. By definition, the bond types L and L' appearing in the atomic rule r must be some L_{w_i}

and L_{w_j}, respectively. Accordingly, using $w_{\text{left},r}$ for the w_i and $w_{\text{right},r}$ for the w_j, we have $(w_{\text{left},r})_{\text{left}_r} \xrightarrow{\alpha_r} (w_{\text{right},r})_{\text{right}_r}$ for the atomic rule r. Without loss of generality, we assume that all the rules R are consistent. (A rule is consistent if it is enabled under some collection.)

For a vector $v \in \mathrm{Z}^t$, we use $v[w_j]$ to indicate the j-th component, for $1 \leq j \leq t$. We construct from R two transitions in VASS G: $q_0 \rightarrow (q_R, v_R^1)$ and $q_R \rightarrow (q_0, v_R^2)$, where q_0 is the initial (and the final) state, and q_R is a new state. Recall that we use I to denote the index set of rule R. That is, $I := \{\text{left}_r : r \in R\} \cup \{\text{right}_r : r \in R\}$. For each $i \in I$, we use $w(i)$ to denote the w with $(w)_i$ appearing in R. Notice that $w(i)$ is unique for each i.

The addition vector v_R^1 for the first transition is the result of the following code:

1. $v := \mathbf{0}$;
2. For each $i \in I$
3. if $w(i) \neq \lambda$ then $v[w(i)] := v[w(i)] - 1$;

The code in above implements the first part of the semantics of the rule R: for each appearance $(w)_i$ (choosing one per i – see the definition of $w(i)$), we "drop" a nonempty bond of type w (line 3) from the collection that R fires upon. In the second part of the the semantics, for the dropped bonds, we transfer objects according the rules and then we "add" those dropped bonds back to the collection, as specified by the addition vector v_R^2 for the second transition, which is the result of the following code:

1. $v := \mathbf{0}$;
2. For each $i \in I$
3. $w := w(i)$;
4. For each $r \in R$
5. if $w(i) = w_{\text{left},r}$ then $w := w - \alpha_r$
   // '$-$' is multiset difference
6. For each $r \in R$
7. if $w(i) = w_{\text{right},r}$ then $w := w + \alpha_r$
   // '$+$' is multiset union
8. $v[w] := v[w] + 1$;
9. $v_R^2 := v$.

□

A subset S of N^n is a *linear set* if there exist vectors $v_0, v_1, ..., v_t$ in N^n such that $S = \{v \mid v = v_0 + a_1 v_1 + \cdots + a_t v_t, \forall 1 \leq i \leq t, a_i \in \mathrm{N}\}$. The vectors v_0 (the constant vector) and $v_1, ..., v_t$ (the periods) are called *generators*. A set S is *semilinear* if it is a finite union of linear sets. The *semilinear reachability* problem for VASS is as follows: Given a VASS G and semilinear sets P and Q, are there vectors $u \in P$ and $v \in Q$ such that $u \rightsquigarrow_G v$? The problem can be easily shown decidable, by making a reduction to the classic (vector to vector) reachability problem (which is known decidable) for VASS.

Lemma 2. *The semilinear reachability problem for VASS is decidable.*

Using Lemma 2 and Theorem 7, it is straightforward to show

Corollary 4. *The bond-type reachability problem for B-bounded RBCS is decidable.*

Notice that, if we remove the condition of B-boundedness from Corollary 4, the bond-type reachability problem is undecidable, as shown in Theorem 5. The undecidability remains even when simple RBCS are considered.

5 Conclusion

In this paper, we introduced Bond Computing Systems (BCS) to model high-level dynamics of pervasive computing systems. The model is a variation of P systems introduced by Gheorghe Paun and, hence, is biologically inspired. We mostly focused on regular bond computing systems (RBCS), where bond types are regular, and study their computation power and verification problems. Among other results, we showed that the computing power of RBCS lies between LBA (linearly bounded automata) and LBC (a form of bounded multicounter machines) and hence, the bond-type reachability problem (given an RBCS, whether there is some initial collection that can reach some collection containing a bond of a given type) is undecidable. We also study a restricted model (i.e., B-boundedness) of RBCS where the reachability problem becomes decidable.

Notice that our model of BCS is not universal. Clearly, if one generalizes the model by allowing rules that can create an object from none, then the model becomes Turing-complete (from the proof of Theorem 4). We are currently implementing a design language based on BCS and study an automatic procedure that synthesizing a pervasive application instance running over a specific network from a BCS specification.

References

1. D. Cook. Health monitoring and assistance to support aging in place. *Journal of Universal Computer Science*, 12(1):15–29, 2006.
2. Z. Dang and O. H. Ibarra. On P systems operating in sequential mode. In *Preproceedings of the 6th Workshop on Descriptional Complexity of Formal Systems (DCFS'04)*, 2004. Report No. 619, Univ. of Western Ontario, London, Canada (2004), 164-177.
3. O. H. Ibarra, H. Yen, and Z. Dang. The power of maximal parallelism in p systems. In *Proceedings of the 8th International Conference on Developments in Language Theory (DLT'04)*, 2004. Lecture Notes in Computer Science, Vol. 3340, pp. 212-224, Springer, 2004.
4. M. Kumar, B. Shirazi, S.K. Das, B.Y. Sung, D. Levine, and M. Singhal. PICO: a middleware framework for pervasive computing. *Pervasive Computing, IEEE*, 2(3):72–79, 2003.
5. L. Lamport and N. Lynch. Distributed computing: models and methods. Handbook of theoretical computer science (vol. B): formal models and semantics, 1157-1199, 1991.
6. C. Martin-Vide, Gh. Paun, J. Pazos, and A. Rodriguez-Paton. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.
7. S. H. Park, S. H. Won, J. B. Lee, and S. W. Kim. Smart home-digitally engineered domestic life. *Personal and Ubiquitous Computing*, 7(3-4):189–196, 2003.
8. Gh. Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
9. Gh. Paun. Introduction to membrane computing. *See P Systems Web Page at http://psystems.disco.unimib.it*, 2004.

10. G.-C. Roman, C.Julien, and J. Payton. A formal treatment of context-awareness. In *Proceedings of FASE'04*. Lecture Notes in Computer Science, Vol. 2984, pp. 12-36, Springer, 2004.
11. W. J. Savitch. A note on multihead automata and context-sensitive languages. *Acta Informatica*, 2(3):249–252, 1973.
12. B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society, Santa Cruz, CA, (1994) 85-90.
13. B. Y. Sung, M. Kumar, B. Shirazi, and S. Kalasapur. A formal framework for community computing. Technical report, 2003.
14. M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.
15. P. Zimmer. A calculus for context-awareness. BRICS Research Series, 2005.