IR Colony Simulator

Šárka Vavrečková[1]

[1]Silesian University in Opava, Faculty of Philosophy and Science, Bezručovo nám. 13, Opava, Czech Republic

Abstract

IR Colonies is a mathematical model designed to simulate communication in networks with Internet of Things devices. It is based on several other theoretical models but has been specifically adapted for this purpose. This paper describes a simulator (in the form of an application) programmed according to the IR Colony model. It outlines the system's communication architecture, object model, and basic aspects of implementation, and also provides a link to download the latest version of the simulator.

Keywords

IR Colony, program, rule, Internet of Things, IR Colony Simulator, Communication Architecture, Object Model

1. Introduction

Membrane computing (introduced by Gheorghe Păun in 1998) is a framework for modelling parallel distributed processing. Mathematical models of membrane systems have been called P Systems. Information about this paradigm is available in [1].

Various models based on P-Systems have appeared over time: one such system is P-Colonies, a simple computational model in which agents share an environment and communicate with other agents and with the environment using programs. P Colonies with transferable programs were introduced in [2]. It is possible to transfer programs between agents and the environment in this type of P Colony, and vice versa. Any agent can thus obtain a program that it will use in subsequent computing steps. A transferable program is a special type of program located in the environment or in an agent and equipped with a condition that must be met for the program to be transferred between the agent and the environment.

Reaction systems were introduced in [3], and in [4, 5] we can find information about networks of reaction systems. Reaction systems are able to use reactants and inhibitors to influence the processes in the system. Reactants can initiate a process, while inhibitors can block it. Networks of reaction systems are graphs with reaction systems as their nodes, the connected reaction systems can communicate with each other using the given network.

IR Colonies were introduced in [6] as a tool designed especially for modelling communication in IoT networks. It is a system based on both the aforementioned P Colonies with transferable programs and reaction systems. IR Colonies take from P Colonies the idea of shared environment and agents equipped with programs, reaction systems were the inspiration for the definition of infrastructure as a graph and the use of the principle of reactants and inhibitors in program rules. In [6] we can also find a sample model of IoT network created using an IR Colony.

Paper [7] analyses the suitability of IR Colonies for simulating communication in a network of IoT devices; specifically, we examined communication models typical of IoT networks and common IoT protocols. The paper compares the communication architecture of the HTTP REST and MQTT protocols, as well as several others, with the capabilities of IR Colonies.

This section is followed by the preliminaries section with a brief introduction to IR Colonies with the full definition of the given system, and the Internet of Things.

Section 3 outlines the simulator design. It specifies the requirements for the resulting system, proposes a communication architecture describing the data flows between the individual parts of the system

ITAT'26: Information technologies – Applications and Theory, September 25–29, 2026, Biele Karpaty, Slovakia

sarka.vavreckova@fpf.slu.cz (Š. Vavrečková)

0009-0003-7121-4743 (Š. Vavrečková)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-09

CEUR-WS.org/Vol-4323/paper30.pdf

(primarily between the main control module and the agents), and presents a simplified object model that mainly shows the classes and the relationships between them.

Section 4 describes the implementation: the choice of programming language, some data structures, and the most important parts of the code, including the system's interaction with the user. It also includes a link to download the project's source files and a detailed example of simulation.

The final section discusses the future development of the project.

2. Preliminaries

We assume the reader to be familiar with the basics of the formal language theory [8] and membrane computing [1].

We denote the length of a word w by $|w|$, and also the number of elements in a set S by $|S|$. The empty word is represented by the symbol ε, so $|\varepsilon| = 0$.

2.1. Objects and Operations in IR Colony

In order to make our IR Colony concept applicable to IoT network simulation, we need objects with *properties* (values). The properties of objects are numbers (e.g. for an object of type temperature, the property is the temperature value, an object specifying the version of a device will have the version number directly as a property). In the basic concept of IR Colony, each object has at most one property and the value of this property is a number, but in principle it is not a problem to modify this concept and use objects with more than one property, moreover of different types (numbers, strings).

The agent state is determined by the set of objects. Because our objects can have a property (or they can have no property), set operations on these sets must be defined a little differently than usual.

Definition 1 (According to [6]). The intersection operation *on sets of objects with properties corresponds to the operation without using properties, except when both sets contain the same object with different property values. The intersection operation is commutative.*

Suppose an object a, and if the given object has a property $x \in \mathbb{R}$, we write it as $a(x)$. Assume that a set cannot contain more than one object of the same type (with the same name), even if it has a different property value. There are the following cases:

1. $\{a, \dots\} \cap \{a, \dots\} = \{a\}$ *and* $\{a(x), \dots\} \cap \{a(x), \dots\} = \{a(x)\}$*, as can be expected,*
2. $\{a(x), \dots\} \cap \{a(y), \dots\} = \emptyset$ *where* $x, y \in \mathbb{R},\ x \neq y$*,*
3. $\{a(x), \dots\} \cap \{a, \dots\} = \{a(x)\}$ *where a in the right-hand set has no property.*

All membership operators (e.g. $\subseteq$) work with properties in the same way.

Definition 2 (According to [6]). The union operation *applied on a set of objects with properties is not commutative. If the same object is in both sets, but with different properties, the object from the set on the right side of the union operator will be the member of the resulting set.*

Suppose an object a and a set M, a is not present in M neither with nor without property. There are the following cases:

1. *if* $a(x) \in M$ *then* $a(x) \in \{a(x)\} \cup M$ *and* $a(x) \in M \cup \{a(x)\}$*,*
2. *if* $a \in M$ *(with no property) then* $a \in \{a \cup M\}$ *and* $a \in M \cup \{a\}$*,*
3. *if* $a \in M$ *then* $a(x) \in \{a(x)\} \cup M$ *and* $a(x) \in M \cup \{a(x)\}$ *(the object with a property takes precedence),*
4. *if* $a(x) \in M$ *then* $a(x) \in \{a\} \cup M$ *and* $a(x) \in M \cup \{a\}$ *(similarly),*
5. *if* $a(x) \in M$ *then* $a(x) \in \{a(y)\} \cup M$ *and* $a(y) \in M \cup \{a(y)\}$ *(the right-hand side property takes precedence).*

For the union operation, the last case listed: This corresponds to updating properties in the right-to-left direction, e.g. $\{a(x), b(y)\} \cup \{b(z), c(r)\} = \{a(x), b(z), c(r)\}$.

If the same object has a property in only one of the sets (the cases 3 and 4), then the result of the union will contain the object with the property. E.g. $\{a(x)\} \cup \{a, b\} = \{a(x), b\}$.

We would assume that two objects of the same type are different if they differ in at least one property.

In some cases, it is necessary to compare the values of properties between two objects, i.e., to use a relational operator. A situation may arise in which one or both of the objects being compared have no property. In that case, the value of the (nonexistent) property of such an object is considered to be the minimum value of the given data type.

2.2. IR Colonies

An IR Colony is a system of agents in a shared environment, connections between agents are defined using a graph. Each agent has its own environment (state) in the form of a set of objects, and objects can also occur in the shared environment.

Definition 3 (IR Colony, [6]). *An IR Colony (IoT Reaction Colony) is a construct* $\Pi_{IR} = (\Sigma, \mathcal{A}, G, \mu, W_0, P_0)$ *where*

- Σ *is a finite nonempty alphabet, a set of base objects, the objects can have default properties,*
- $\mathcal{A} = \{A_1, \ldots, A_m\}$ *is a finite nonempty set of agents,*
- $G = (V, E)$ *is a graph,* V *is a finite set of nodes,* $|V| = m$, E *is a finite set of edges,*
- $\mu\colon V \to \mathcal{A}$ *is a location function, locating agents into the graph nodes,*
- $W_0 \subseteq \Sigma,\ W_0 \neq \emptyset$ *is the initial state of the shared environment,*
- P_0 *is the initial set of programs located in the shared environment.*

If the graph G *is undirected, then it is connected. If* G *is directed, then it is weakly connected.*

An agent $A_j,\ 1 \leq j \leq m$, *is a pair* $A_j = (w_j, P_j)$ *where*

- $w_j \subset \Sigma,\ w_j \neq \emptyset$ *is the initial state of the agent,*
- P_j *is the initial set of programs of the given agent,* P_j *is finite, and can be empty.*

The symbols (objects) in rules may or may not have properties specified. Listing a property on the left side of a rule indicates a conditional application of that rule. The properties are numbers from $\mathbb{R}$. The rules *can be in one of the following forms:*

- *evolution rule:* $a \to b$ *or* $a(x) \to b(y),\ a, b \in \Sigma$, *an agent evolves its state,* $x, y \in \mathbb{R}$ *are properties of the given objects,*
- *deletion rule:* $a \to \varepsilon,\ a \in \Sigma$,
- *multicast rule:* $a \xrightarrow{\text{out}} b,\ a, b \in \Sigma$ *to send the object* b *to all outgoing edges,* a *remains inside the sending agent; if* $a = b$, *it is not necessary to specify the properties of the objects, the current property of* a *is used,*
- *backup rule:* $a \to,\ a \in \Sigma$, *to put the object down into the environment (including its current property), the object* a *remains in the agent's state,*
- *restoration rule:* $\leftarrow b,\ b \in \Sigma$, *to pick an object up from the environment (including its current property), the object* b *remains in the environment; if* b *has been present in the state of the given agent, the parameter will be rewritten (updated),*
- *programming rule:* $p \triangleright$, *the format is explained below at the end of the definition.*

The backup and restoration rule types apply the union operation, including the processing of parameters. Denote $\mathcal{U}$ *the set of all possible rules for* Π_{IR}. *A* program *in* Π_{IR} *is a construct* $p_{\text{lab}} = (\text{lab}, U, R, I)$ *where*

- *each program has the own unique label* lab,
- $U \subseteq \mathcal{U}$ *is a finite nonempty set of rules,*

- $R \subseteq \Sigma \cup \mathcal{U}$ *is a finite set of reactants, reactants can be:*
  - *an object with or without a property (a relational expression can be added to the object for its property),*
  - *a relation between properties of different objects,*
  - *a program,*
- $I \subseteq \Sigma \cup \mathcal{U}$ *is a finite set of inhibitors,* $R \cap I = \emptyset$*, the syntax of elements of the set is the same as for* R*.*

The set U *must be deterministic in the sense that the same object must not appear on the left-hand side of any two rules.*

A programming rule *is a construct* $(\mathrm{label}, U, R, I) \triangleright$ *with the parts of this sequence of the same meaning as in the definition of a program. The programming rule creates a new program with the given parameters and stores it in the rule repository in the environment.*

Definition 4 (Application of rules and programs, [6]). *Assume an IR Colony* $\Pi_{IR} = (\Sigma, \mathcal{A}, G, \mu, W_0, P_0)$ *and* $G = (V, E)$ *with the set of agents* $\mathcal{A} = \{A_1, \ldots, A_m\}$*.*

A state of a given agent $A_j,\ 1 \leq j \leq m$ *is a set of objects, the objects have or have not properties.*

A rule $r \in U$ *with an object* $a \in \Sigma$ *on the left side* is applicable *to a state* W_j *of a given agent* A_j*, iff the object* a *is present in* W_j *including potential parameters. The restoration rule* $\leftarrow b$ *is applicable iff the object on the right side is present in the environment. The programming rule is always applicable.*

A program p_{lab} *in an agent* $A_j = (W_j, P_j)$ is applicable *iff*

- *all rules in* U *are applicable to* W_j*,*
- $(R \cap \Sigma) \subseteq W_j$*,* $(R \cap \mathcal{U}) \subseteq P_j$*,*
- $I \cap W_j = \emptyset$*,* $I \cap P_j = \emptyset$*.*

During a single step*, the following happens (for a* j^{th} *agent):*

1. *the agent checks the program repository and updates its own programs: if there is a program with a label belonging to one of the agent's programs, the agent's original program is overwritten by a new program with the same label located in the repository,*
2. *the agent nondeterministically chooses one of its applicable programs,*
3. *all rules in the program are processed with influencing the own state and computation of the result and context sets for the given agent and step,*
4. *the result sets are used to calculate the new state of agents.*

2.3. Internet of Things

IoT (Internet of Things) devices are usually small devices or simply sensors connected to network, and the main feature that distinguishes an IoT network from conventional computer networks is the ability of devices to communicate with one another automatically. The goal is to allow devices to respond on their own to changes in the environment by altering their state and, if necessary, changing their behavior. Within the network, there are devices that serve as data sources (sensors, such as a thermometer, pressure sensor, camera with image recognition, etc.) and devices that serve as data destinations (a doorbell, a window-opening control mechanism, a display, an alarm, etc.). The first type of device sends data, while the second type receives data and reacts to it.

In [9] several definitions of IoT network can be found, a more compact definition is provided in [10]. We can compose the following definition from them:

Definition 5 ([9], [7]). *The Internet of Things (IoT) is a network of various types of smart objects (so called things) and devices. The things are connected to the Internet and communicate with each other with minimum human interface. They are embedded with abilities as sensing, analyzing, processing and self-management based on interoperable communication protocols and specific criteria. These smart things should have unique identities and personalities.*

IoT networks operate using a wide range of protocols; we are interested in the protocols used for communication between devices. Since battery-powered small sensors are common in IoT networks, IoT devices are expected to operate in an energy-efficient manner, a factor that these protocols should also take into account. For this reason, the protocols used in IoT networks differ slightly from those in conventional networks, or there are modified, more energy-efficient versions of them.

There are several *communication models* (communication architectures) [9] used in the IoT world. Two of these models are the best known: Request-Response (Figure 1) and Publish-Subscribe (Figure 2).

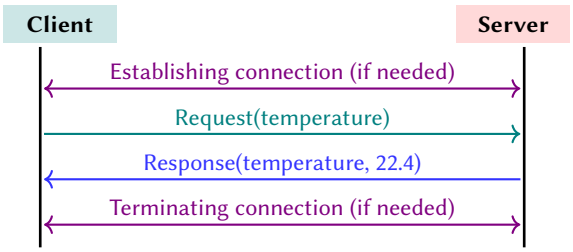


Figure 1: Example of communication in the Request-Response communication model[7]

The Request-Response model is similar to the communication model of a typical computer network and does not require any special type of device; there are simply devices that request data and devices that respond to those requests.

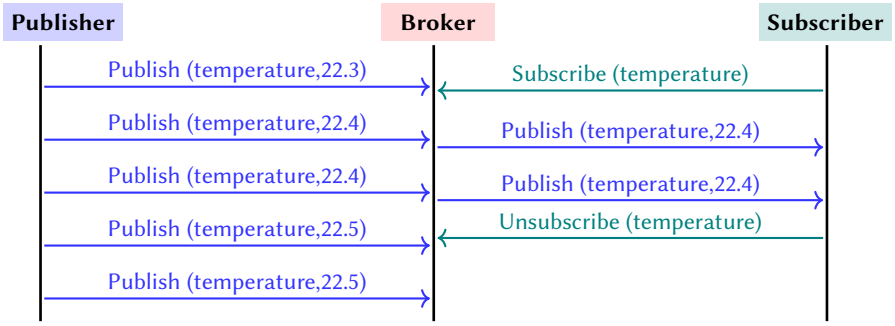


Figure 2: Example of communication in the Publish-Subscribe communication model[7]

The Publish-Subscribe model typically involves a special type of device called Controller, Broker, or simply an intermediary device. The Broker accepts subscriptions to data streams (similar to journal subscriptions), receives data from publisher-type devices, and distributes it to subscribers according to the subscriptions it has received.

A typical example of the Request-Response model is the HTTP REST protocol, while a typical example of the Publish-Subscribe model is the MQTT protocol. An analysis of the possible relationship between the IR Colony concept and the HTTP REST and MQTT protocols—or, more specifically, the corresponding communication models—can be found in [7].

3. IR Colony Simulator Design

Our goal is to create a communication simulator for an IoT network that could be used, for example, when designing a new IoT network, simulating data flows in such a network, or optimizing an existing design.

The first step was to create a mathematical model describing such communication (applicable to both Request-Response and Publish-Subscribe communication architectures)—that is, IR Colony is defined and analyzed in [6] and [7].

This paper covers the second step: converting the mathematical model into code to create a functional simulator that includes agents and the relationships between them. Each agent has its own internal

environment (state) and set of programs, and the agents can influence one another by exchanging data (objects with properties).

The possibilities for the next steps are discussed at the end of this paper.

3.1. Requirements

The first version of the IR Colony Simulator application should cover most of the functionality of the theoretical IR Colony model:

- agents, their programs with various types of rules and conditional execution using reactants and inhibitors,
- the ability to evolve an agent's state (evolution and deletion rules),
- interconnections between agents (unidirectional) and their use via multicast-type rules,
- the entire system environment and the ability to use it through backup and restoration rules,
- a synchronization mechanism that determines the intervals at which agents are to operate.

For debugging purposes, a basic text-based interface is sufficient for now, with the ability to start and stop the simulation, list the status of agents and (virtual) connected devices, and enable verbose mode, which displays the rules currently in use.

Objects are uniquely identified by their names, which are of type string (we chose string over char because it is more illustrative and makes the logs describing the calculation easier to read). For now, the object properties can be of type int; later, rational numbers or a variant type can be simply implemented.

3.2. Communication Architecture

The system assumes the existence of a central component; alternatively, the main control module may be part of one of the connected devices. The *main control module* serves primarily as a storage location for the system environment and a program repository.

An agent can save a copy of an object from its state to the system environment (the *backup* rule) and retrieve it at any time later (the *restoration* rule). And since this is essentially a shared environment, it is also one of the ways agents can exchange objects on a global level.

Multicast rules are used to send objects to selected agents. In the IR Colony as the mathematical model, paths for this type of communication are defined using a backbone graph; in the application, each agent simply has a list of agents that are to be the recipients of its multicast messages. The sources of these messages are typically simple sensors (thermometers, moisture meters, etc.), so it will likely not be necessary to have separate lists of recipients for different rules—a single list is sufficient for each agent.

Figure 3 illustrates the use of several types of rules for communication between system components. There are also rules designed to govern the evolution of the state (the internal environment) of the agent itself: *evolution* and *deletion* rules.

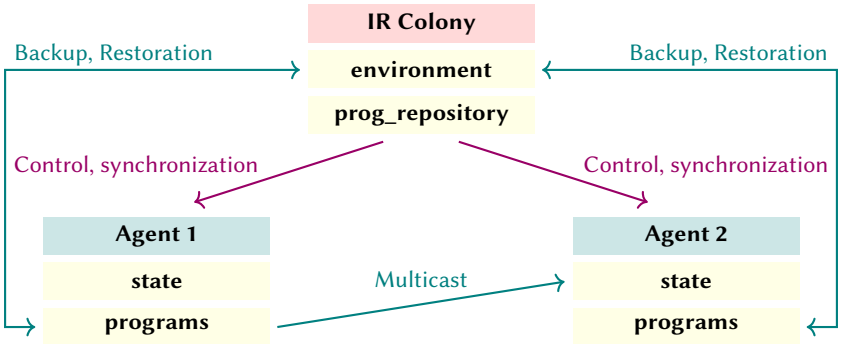


Figure 3: Communication architecture, communication between system parts

The program repository is a central storage location for programs. Although this repository currently exists, it is not yet fully functional. However, in the next phase, it will primarily serve as the equivalent of an update server. Any agent will be able to upload a program to the repository using the *programming* rule, and agents will periodically (for example, before executing each step) check whether there is a newer version of their programs in the repository. If so, they can download the new version. The paper [6] explains how this might work in practice using an example.

3.3. Structure Model

Communication will take place using objects, so we need a class for objects: *IRObject*. We must take into account the fact that objects may or may not have a property (an assigned value) and it must be possible to verify whether an object has a defined value or not. Since we will often need to manipulate sets of objects, the *IRSet* class has also been created for this purpose.

Reactants and inhibitors are handled in the same way, so we can have a single common class for them: *Effector*. We also have the *EffectorSet* class, which comes useful for effectors. Effectors can be of various types (for example, they verify whether a certain element in the agent's environment has a specific value, or whether its value is in a certain relation to a specific value or to the value of another object in the environment). Therefore, we store the relation and the names of one or two objects. Objects with values are not directly involved here; in effectors, we are only interested in the names of the objects, and we determine the values from the agent's environment.

For effectors, we will often check whether they relate to a specific set of objects; we have created a function for this purpose.

Figure 4 shows a simplified model in the form of classes and the relationships between them. The figure primarily illustrates the relationships where elements of one class belong to another class, but for clarity, we have also included a list of the most important methods of some classes.

Let us move on to more complex classes. There is a class called *ProgRule* for program rules, which, in addition to the rule type, specifies the objects with which the rule is to operate. Some rules require two

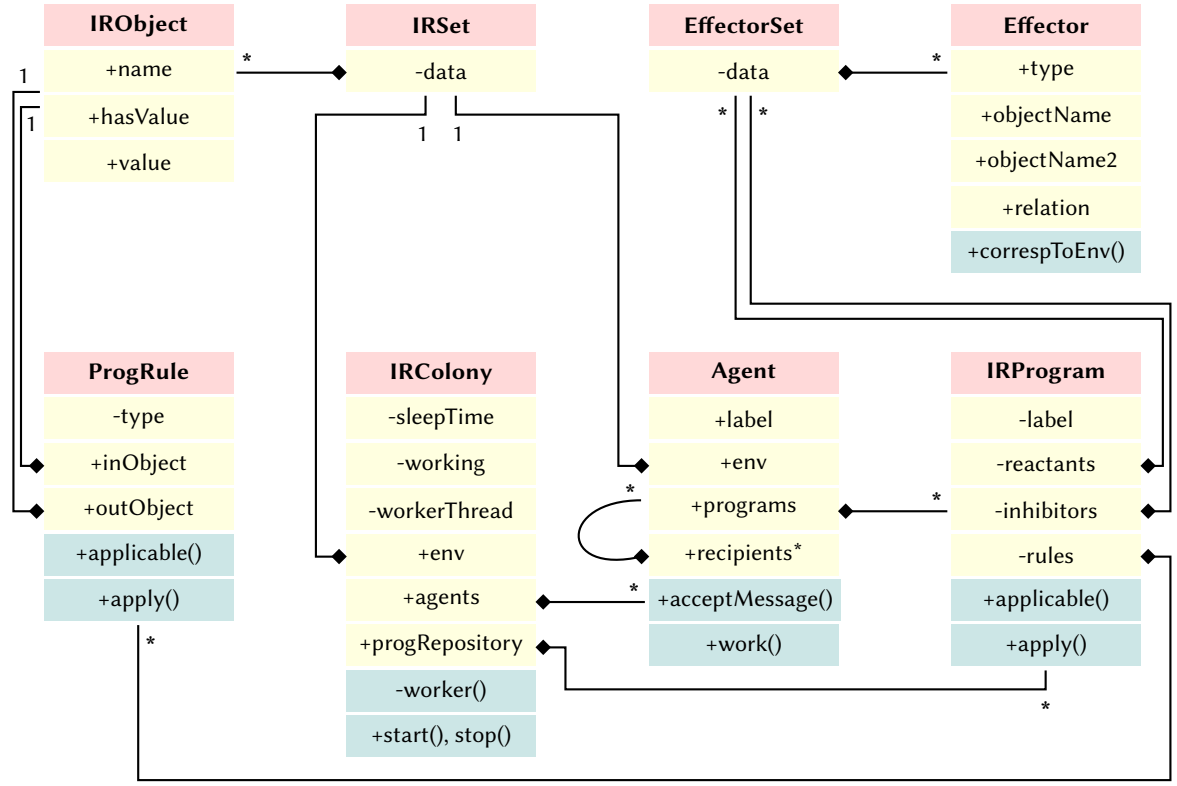


Figure 4: Simplified class model of the application, relationships between classes

objects, others just one; here, there is space for both objects on the left and right sides of the rule.

Each program (*IRProgram*) consists of rules (a set of elements of the ProgRule type) and sets of reactants and inhibitors (*EffectorSet*).

The class for agents (*Agent*) is equipped with an environment (*IRSet*), a set of programs (*IRProgram*), and a set of pointers to agents that are to be the recipients of multicast communication.

The *IRColony* class has its own environment and set of agents, and there is also a program repository available here.

As we can see, the classes for rules and programs have functions for verifying applicability and for applying the rule or program itself. Each agent has a function for the computation step called `work()`. IR Colony is equipped with the `worker()` function for the worker thread, in which the `work()` functions of individual agents are called periodically.

4. Implementation

4.1. Technologies

I considered implementing the system in Python, Java, or C++. Python was found to be impractical, particularly with regard to function parameters, due to its dynamic typing; furthermore, its lower performance could become an issue in the future when a separate thread will be required for each agent. Java, as a purely object-oriented language running on the JVM virtual machine, requires a larger amount of code and is not suitable for low-level programming; due to the use of the garbage collector, there isn't much flexibility for potential memory optimization, and while its performance is better than Python's, it is worse than C++'s.

The C++ language was therefore chosen for its better performance (including when using multiple threads) and greater control over memory. Neither of these features is particularly important in this phase, but that will change in later phases. The object-oriented model of C++ is sufficient for our purposes, especially in the newer language specifications. A graphical user interface is also planned for one of the later phases, which should not be a problem either.

Since this is a simulator of communication between IoT devices, the agents should be linked in some way to the corresponding IoT devices. We are not addressing this at this phase, but it is assumed that the state of the device is reflected in the values of the objects within the agent's environment.

4.2. Data Structures and Code

The project source code is currently available at https://github.com/sarkavavreckova/IRColSimulator/. The code is currently at a stage where it can even be run in cloud compilers such as https://www.onlinegdb.com/, so there is no need to install anything locally.

The classes IRObject and Effector look as follows:

```
class IRObject {
public:
    string name;
    bool hasValue;
    int value;
    ...
    void print() const;
    string getAsString() const;
 };

enum EffectorType { NONE, OBJECT, OBJECTWITHREL, RELBETWEENOBJECTS, PROG };
enum EffectorRelation { NONEREL, EQUAL, LESS, GREATER };    // nothing, ==, <, >

class Effector {
public:
    EffectorType type;
    string objectName;          // for OBJECT, OBJECTWITHREL, RELBETWEENOBJECTS
```

```
    string objectName2;          // for RELBETWEENOBJECTS
    EffectorRelation relation;   // for OBJECTWITHREL, RELBETWEENOBJECTS
    int relValue;                // for OBJECTWITHREL
    ...
};
```

The IRSet class encapsulates a set of objects. However, instead of the `set` type, the `map` type was chosen, as it allows for more flexible manipulation of the contained objects. In contrast, the `set` type proved to be suitable for the set of effectors.

```
class IRSet {
private:
    map<string,IRObject> data;
...
```

```
class EffectorSet {
private:
    set<Effector> data;
...
```

The definitions of all the classes mentioned in the section on the object model can be found in files with the .h extension.

According to the IR Colony definition, each agent executes one (applicable) program per step. Although it could make a nondeterministic choice if there are multiple applicable programs, for the sake of simplicity, it always runs the first applicable program it finds.

```
void Agent::work(IRSet& IREnv, set<IRProgram>& repoz) {
    for (auto itr = programs.begin(); itr != programs.end(); ++itr) {
        if (itr->applicable(env, IREnv)) {
            itr->apply(label, env, IREnv, recipients);
            break;
        }
    }
}
```

The applicability of the program is determined by verifying included rules (some rules are always applicable, while others are applicable only when a specific object is located in the agent's environment or in the system's environment) and by verifying that the conditions specified by reactants and inhibitors are met.

If the program is applicable, all of its rules are executed sequentially. The most complex rule to execute is the *multicast* rule, because it is necessary to specify which object and with what value will be transported, and then the transport is carried out.

```
void ProgRule::apply(const string agentLabel, const string progLabel, IRSet& agentEnv,
    IRSet& IREnv, set<Agent*>& recip) const {
    switch(type) {
    ...
    case MULTICAST: {
        IRObject objectToSend;
        if (outObject.hasValue)
            objectToSend = outObject;
        else if (inObject.hasValue) {
            objectToSend.name = outObject.name;
            objectToSend.hasValue = true;
            objectToSend.value = inObject.value;
        }
        else {
            objectToSend = outObject;
            objectToSend.hasValue = true;
            auto itr= agentEnv.findName(inObject.name);
            objectToSend.value = itr->second.value;
        }
        for (auto itr = recip.begin(); itr != recip.end(); ++itr) {
            (**itr).acceptMessage(objectToSend);
        }
        if (verboseMode)
            cout << endl << "Agent " << agentLabel << ",\tprogram " << progLabel << ",\trule "
            << ruleTypeToString(type) << ":\t" << objectToSend.getAsString() << " ->out";
```

```
        break;
    }
    ...
```

For security reasons, receipt of the message is confirmed by the recipient (function `acceptMessage()`).

The code also shows the agent's behavior (or rather, the rules in its program) when "verbose mode" is enabled: the simulator then behaves similarly to a device with logging enabled, meaning it outputs the applied rules to the console, including information about the agent, the program, and the processed objects.

The verbose output might look something like this:

```
Agent ControlPanel,     program cLight,         rule MULTICAST: lbState(0) ->out
Agent Thermometer,      program tempOut,        rule MULTICAST: temp(24) ->out
Agent WindowControl,    program wOpenThermo,    rule EVOLUTION: wState(1) -> wState(1)
```

The current version of the simulator uses two threads: the main thread handles communication with the user, and the computation thread performs the calculations once the simulation starts. In the future, each agent will have its own computation thread.

```
void IRColony::start() {
    working = true;
    workerThread = thread(&IRColony::worker, this);
}
```

When the simulation starts, the `IRColony::worker()` function is called inside the worker thread, which iterates through all agents at regular intervals; each agent performs a calculation step (if it finds an applicable program):

```
void IRColony::worker() {
    while(working) {
        for (auto& [label, agent] : agents) {
            agent.work(env, progRepository);
        }
        this_thread::sleep_for(chrono::seconds(sleepTime));
    }
}
```

4.3. Communication

For testing purposes, four agents (smart light bulb, control panel, window, and thermostat) are hard-coded into the project's main function. In later versions, it will be possible to add agents using a JSON file or, once a graphical interface is implemented, by "clicking".

The current version communicates via a simple text menu. It allows the user to display information about the agents, display the state of the devices (which reflects the state of the agents' environment), increase or decrease the temperature on the thermostat, start and stop the simulation, enable or disable verbose mode, and, of course, exit the program.

```
*******************************************************************************
Press one of the following keys (and ENTER), you can use both uppercase/lowercase:

P ... Print agents info
S ... Start (start IRColony)
E ... End (stop working)
D ... Print state of devices
+ ... Increase temperature
- ... Decrease temperature
V ... Verbose mode (undo: T)
T ... Silent mode
Q ... Quit this program
*******************************************************************************
Your choice:
```

The ability to increase and decrease the temperature represents the functionality of connecting a real IoT device to an agent—though, for now, only symbolically.

4.4. Detailed Simulation Example

The application currently features a small sample IoT network with four agents connected to IoT devices. The corresponding IR Colony is as follows:

The simplest agent is a smart light bulb. There are two objects in its state: the device state (0 for lights off, 1 for lights on), and the light intensity (0–12). There are no programs inside this agent.
$A_{LightBulb} = (\{lbIntensity(8), lbState(0)\}, \emptyset)$

The corresponding agent in the simulator is

```
Agent LightBulb:
Environment (2 objects):        lbIntensity(8)  lbState(0)
Programs (0 programs):
```

The second agent represents a thermometer. Its object $temp$ is intended to store the current temperature. There is only one program in the set of programs, with a single (multicast) rule.
$A_{Thermometer} = (\{temp(24)\}, \{\text{tempOut}\})$ with the program:
$(\text{tempOut}, \{temp \xrightarrow{\text{out}} temp\}, \emptyset, \emptyset)$

The corresponding agent in the simulator is

```
Agent Thermometer:
Environment (1 objects):        temp(24)

Programs (1 programs):
** Program tempOut
Rules:
(MULTICAST: temp ->out temp)
Reactants (0 effectors):
Inhibitors (0 effectors):
```

Next, we have an agent acting as a control panel for the manual handling of devices. It needs the objects with the current state of the controlled devices (the light bulb, the window), and one additional object indicating manual handling for the window. This agent has three multicast rules that notify connected components of changes to a given value.
$A_{ControlPanel} = (\{cLightState(0), cWinManual(0), cWinState(0)\},$
$\{\text{cLight}, \text{cWindowManual}, \text{cWindowState}\})$

$(\text{cLight}, \{cLightState \xrightarrow{\text{out}} lbState\}, \emptyset, \emptyset)$
$(\text{cWindowManual}, \{cWinManual \xrightarrow{\text{out}} wManual\}, \emptyset, \emptyset)$
$(\text{cWindowState}, \{cWinState \xrightarrow{\text{out}} wState\}, \{cWinManual(1)\}, \emptyset)$

The corresponding agent in the simulator is

```
Agent ControlPanel:
Environment (3 objects):        cLightState(0)  cWinManual(0)   cWinState(0)

Programs (3 programs):

** Program cLight
Rules:
(MULTICAST: cLightState ->out lbState)
Reactants (0 effectors):
Inhibitors (0 effectors):

** Program cWindowManual
Rules:
(MULTICAST: cWinManual ->out wManual)
Reactants (0 effectors):
Inhibitors (0 effectors):
```

```
** Program cWindowState
Rules:
(MULTICAST: cWinState ->out wState)
Reactants (1 effectors):        OBJECTWITHREL cWinManual(1)
Inhibitors (0 effectors):
```

The role of the following agent is a window control, and it is necessary to synchronize inputs from three different sources: the thermometer, the CO_2 sensor, and the manual operation on the control panel. The programs are not triggered when the manual mode (operation from the control panel) is enabled.

$A_{WindowControl} = (\{wState(0), wManual(0)\},\ \{\text{wOpenThermo, wOpenCO2, wClose}\})$
$(\text{wOpenThermo}, \{wState(0) \to wState(1)\}, \{temp(> 22)\}, \{wManual(1)\})$
$(\text{wOpenCO2}, \{wState(0) \to wState(1)\}, \{sCO2State(> 1200)\}, \{wManual(1)\})$
$(\text{wClose}, \{wState(1) \to wState(0)\}, \emptyset, \{wManual(1), temp(> 22), sCO2State(> 1200)\})$

For simplicity, the agent connected to the CO_2 sensor has been omitted here; therefore, the presence of related objects and programs is for illustrative purposes only.

The corresponding agent in the simulator is

```
Agent WindowControl:
Environment (2 objects):        wManual(0)      wState(0)

Programs (3 programs):

** Program wClose
Rules:
(EVOLUTION: wState(1) -> wState(0))
Reactant (0 effectors):
Inhibitors (3 effectors):       OBJECTWITHREL sCO2State(>1200)  OBJECTWITHREL temp(>22)
    OBJECTWITHREL wManual(1)

** Program wOpenCO2
Rules:
(EVOLUTION: wState(0) -> wState(1))
Reactants (1 effectors):        OBJECTWITHREL sCO2State(>1200)
Inhibitors (1 effectors):       OBJECTWITHREL wManual(1)

** Program wOpenThermo
Rules:
(EVOLUTION: wState(0) -> wState(1))
Reactants (1 effectors):        OBJECTWITHREL temp(>22)
Inhibitors (1 effectors):       OBJECTWITHREL wManual(1)
```

At the start of the calculation, the ‘temp‘ object is not present in the agent’s environment, even though it appears in the rules. This is correct; this object is inserted into the agent’s environment in the first step of the calculation by the Thermometer agent using a multicast rule.

The IR Colony is defined as follows:
$\Pi_{IR} = (\Sigma, \mathcal{A}, G, \mu, W_0, P_0)$ where Σ contains all of the objects listed above; set A contains the four agents described; we do not yet need any objects in the environment; and the set of environment rules P_0 is also empty (it would only be relevant upon full implementation of the PROGRAMMING rules).

All that remains is to define the connections between agents. In IR Colony, we would specify a graph G and a mapping μ; in the simulator, we will define the relationships using the following commands:

```
ircolony.addLinkBetweenAgents("ControlPanel","LightBulb");      // ControlPanel ---> LighBulb
ircolony.addLinkBetweenAgents("ControlPanel","WindowControl"); // ControlPanel ---> WindowControl
ircolony.addLinkBetweenAgents("Thermometer","WindowControl");  // Thermometer  ---> WindowControl
```

This information also appears in the list of agent properties; e.g. for the ControlPanel agent:

```
Recipients of multicast messages:    LightBulb     WindowControl
```

We can check the device's initial state (by pressing the "D" key):

```
ControlPanel:   cLightState(0)  cWinManual(0)   cWinState(0)
LightBulb:      lbIntensity(8)  lbState(0)
Thermometer:    temp(24)
WindowControl:  wManual(0)      wState(0)
```

For now, the window is closed (`wState = 0`), even though the thermometer reads 24, because the simulation has not yet been started. Let's run the IR Colony defined above in the simulator (using the "S" key) and display the device status again:

```
ControlPanel:   cLightState(0)  cWinManual(0)   cWinState(0)
LightBulb:      lbIntensity(8)  lbState(0)
Thermometer:    temp(24)
WindowControl:  lbState(0)      temp(24)        wManual(0)      wState(1)
```

As we can see, an object containing temperature information has been sent to the WindowControl agent, and the window has opened (`wState = 1`).

Let's return to the theoretical model. In each step of the calculation (including the first one), the agent $A_{Thermometer}$ is primarily active, as well as $A_{ControlPanel}$. $A_{ControlPanel}$ nondeterministically selects one of the applicable programs and runs it. The presence of nondeterminism here isn't entirely logical: it would be better to adjust the agent's configuration so that it sends changes to the network only when the user selects a change to a parameter (this could be addressed, for example, by adding reactants that indicate a change has occurred in the device). The problem of nondeterminism in the simulator will be addressed in a later version; for now, the first applicable rule is beeing selected.

The $A_{Thermometer}$ agent regularly sends the $temp$ object, including its value, to connected agents using the program $(\text{tempOut}, \{temp \xrightarrow{\text{out}} temp\}, \emptyset, \emptyset)$. In the simulator, the corresponding agent behaves the same way.

We press the "-" key twice to lower the temperature by two degrees to 22. Consequently, the window should close—that is, the wClose program should be executed by the WindowControl agent. Let's verify the result by displaying the device status (the "D" key):

```
WindowControl:  lbState(0)      temp(22)        wManual(0)      wState(0)
```

The value of the `wState` object is 0, so the window has been closed.

The ability to demonstrate that the calculation process in the theoretical model and the simulation in the simulator correspond is limited, although the theoretical model also assumes the existence and active participation of devices that are connected to the agents. In the theoretical model, we would also need to reduce the value of the $temp$ object in agent $A_{Thermometer}$ by 2 degrees; then the agent $A_{WindowControl}$ behaved the same way, its program wClose would be applicable.

5. Discussion and Conclusion

The goal was to create the first version of an IoT network simulator that would implement most of the functionality of the IR Colony mathematical model. In previous papers, this mathematical model was described and its compatibility with common communication protocols for the Internet of Things was analyzed; now, the implementation phase is beginning. The project is already operational, although control options are currently limited, and the functionality for updating programs via the *programming* rule is not yet fully supported.

Now is a good time to start a discussion about the simulator's inner mechanism design. If certain parts of the code or the communication structure can be optimized, it is better to do so now rather than once the project grows in scope.

A certain obstacle to implementation is the requirement for nondeterminism in selecting one of the agent's applicable programs. This requirement was carried over into the IR Colony definition from the P Colony model, but it is not practical for real-world deployment. In this version of the simulator, the selection of an applicable program is currently handled sequentially (the first applicable program is

selected); however, this may result in some applicable programs not being executed at all. Another possibilities would be (1) random selection, (2) pseudo-parallel execution of all applicable programs, or (3) establishing conditions for the mutual exclusion of programs (e.g. using inhibitors). We will address this issue in one of the upcoming versions of the simulator.

In each step, all agents that find an applicable program are active. If an agent applies a *programming* rule, it sends a new version of a program to the repository in the environment, where another agent retrieves it. A program updated in this way would not be usable until the next step. The theoretical model already takes this into account: at the beginning of each computation step, every agent checks the program repository and downloads any updates if necessary. The new version of the program is therefore used in a different step than the one in which it was created. Although the simulator does not yet have full built-in support for *programming* rules, the repository and part of the mechanism already exist, and program updates will work the same way as in the theoretical model.

In the future, the simulator's functionality will be expanded to include full support for *programming* rules and to resolve the described problem of nondeterminism, the importing of agents will be resolved (by the JSON format), user interaction will be generally improved, and options for fully parallel agent execution with synchronization will be explored. We are also beginning to consider options that directly support certain IoT protocols.

Declaration on Generative AI

During the preparation of this work, the author used Grammar Checker (https://www.grammarcheck.net/editor/) in order to: Grammar and spelling check. After using this service, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] G. Păun, G. Rozenberg, A. Salomaa, The Oxford Handbook of Membrane Computing, Oxford University Press, New York, 2010.

[2] L. Ciencialová, L. Cienciala, Transferable Knowledge in P Colonies, in: Information Technologies – Applications and Theory 2022 (ITAT 2022), volume 3226, Zuberec, Slovakia, 2022, pp. 167–174.

[3] A. Ehrenfeucht, G. Rozenberg, Reaction Systems, Fundam. Informaticae 75 (2007) 263–280. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi75-1-4-15.

[4] P. Bottoni, A. Labella, G. Rozenberg, Networks of Reaction Systems, International Journal of Foundations of Computer Science 31 (2020) 53–71. doi:10.1142/S0129054120400043.

[5] B. Aman, From Networks of Reaction Systems to Communicating Reaction Systems and Back, in: J. Durand-Lose, G. Vaszil (Eds.), Machines, Computations, and Universality, Springer International Publishing, Cham, 2022, pp. 42–57. doi:10.1007/978-3-031-13502-6_3.

[6] Š. Vavrečková, Transferable programs and reactions for modeling iot network, in: CEUR Proceedings of the 24rd Conference Information Technologies – Applications and Theory (ITAT 2024), 2024, pp. 111–120.

[7] Š. Vavrečková, Compatibility of IR Colonies and Protocols for Communication between IoT Devices, in: CEUR Proceedings of the 21st Conference Information Technologies – Applications and Theory (ITAT 2025), 2025, pp. 264–279.

[8] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, 1979.

[9] W. Kassab, K. A. Darabkh, A–Z Survey of Internet of Things: Architectures, Protocols, Applications, Recent Advances, Future Directions and Recommendations, Journal of Network and Computer Applications 163 (2020). doi:10.1016/j.jnca.2020.102663.

[10] Š. Vavrečková, Membrane System as a Communication Interface between IoT devices, in: CEUR Proceedings of the 22nd Conference Information Technologies – Applications and Theory (ITAT 2022), 2022, pp. 184–190.