Equivalence of Processes in Environments with Commutative Objects

Hayk Grigoryan
Yerevan State University
Yerevan, Armenia
e-mail: hgrigorian@gmail.com

Arsen Shoukourian
Institute of Informatics and Automation Problems
Yerevan, Armenia
e-mail: arsen.shoukourian@gmail.com

ABSTRACT

The equivalence of processes is an important constituent of process optimization. The article considers the functional equivalence problem for processes in object-oriented environments for the case when the turn of executing operations for different objects is not essential. It is shown that this problem can be reduced to the equivalence problem of regular expressions over a partially commutative alphabet.

Keywords

process, equivalence, multitape automata, multidimensional automata, object-oriented environment

The equivalence of processes, important for process optimization, is considered. A model defined over an object-oriented environment [1, 2] is extended and formalized to reflect the introduced problem of functional equivalence of processes within the mentioned model. The extension includes the definition of commutative objects as well as an extension of formal semantics for the considered cases. The problem of functional equivalence is formulated and it is shown that this problem can be reduced to the equivalence problem of regular expressions over a partially commutative alphabet, which is solvable [3].

A formal model of an environment based on a finite set of communicating objects is introduced below. Objects are communicating with each other via a finite set of messages. As there could be several messages passed to a given object, each object has a possibility to gather a finite number of received input messages in a queue before processing. A newly received message is ignored, if it comes after the message queue is filled completely.

Object consists of a unique identifier, a finite set of states and a finite set of operations. Operations are carried out in response to messages, last some time interval, and, for a current state, result in a new state as well as in a vector of messages to be sent to other objects. A state of the environment is a vector of states of contained objects. State and operation sets of different objects within the environment are considered disjoint.

Basic model equations describing environment, object, operation as well as duration of operations and finite message queues are defined as follows:

Environment = $(o_1, \ldots, o_n)$, where $o_i \in$ {Object}, $i=1,\ldots,n$

Object = (ID, {State}, {Operation})

MSG = $\{msg_e, msg_1, \ldots, msg_k\}$

Operation = {State} × MSG → {State} × {Reaction}

Reaction ∈ MSG^n, Reaction [i] is the component i of the vector Reaction, $i=1,\ldots,n$

Duration = {Operation} → {1, 2, …}

Then the next group of model equations is added to describe message communication within the environment, including message addition to queues for objects of the environment. Each object queue has its own length p_i.

EnvState = $(s_1, \ldots, s_n)$, $s_i \in$ {State} for the object o_i, $i=1,\ldots,n$, EnvState [i] = s_i

$$\text{MSGQueue}^{(p)} = \bigcup_{m=0}^{p} (\text{MSG} - msg_e)^m,\ p \in \{0, 1, 2, \ldots\},$$

$\text{MSGQueue}^{(p)}[i]$ is the component i of the vector $\text{MSGQueue}^{(p)}$

$$\text{EnvInput} \in \text{MSGQueue}^{(p_1)} \times \ldots \times \text{MSGQueue}^{(p_n)}$$

Finally a notion of scene is introduced, which describes the status of the environment at a given time moment t as well as the change of EnvInput basing on a given Reaction.

Scene = (t, EnvState, EnvInput), t ∈{1, 2, …}

EnvInput ♀ Reaction = (EnvInput[1] ♀ Reaction[1], …, EnvInput[n] ♀ Reaction[n])

$\text{MSGQueue}^{(p)} = (msg^{(1)}, \ldots, msg^{(m)})$, where $msg^{(1)}, \ldots, msg^{(m)} \in (\text{MSG} - msg_e)$, $m \le p$

$\text{MSGQueue}^{(p)}$ ♀ msg = $(msg^{(1)}, \ldots, msg^{(m)}, msg)$, if $m<p$, $msg \in (\text{MSG} - msg_e)$

$\text{MSGQueue}^{(p)}$ ♀ msg = $\text{MSGQueue}^{(p)}$, if m=p

$\text{MSGQueue}^{(p)}$ ♀ msg_e = $\text{MSGQueue}^{(p)}$

Any operation op of a given object o can be naturally extended to operation $op^{(ext)}$: {EnvState} × {EnvInput} → {EnvState} × {EnvInput} in the following way.

∀ EnvState ∀ Env$op^{(ext)}$ (EnvState, EnvInput) = (EnvState′, EnvInput′), where

EnvState′[j] = EnvState [j], j ≠ i

EnvState′[i] = (op (EnvState[i], (EnvInput [i]) [1]))[1]

EnvInput′ = EnvInput ♀ (op (EnvState [i], (EnvInput [i])[1]))[2]

where (EnvInput [i])[1] is the first message in the queue EnvInput[i], (op (EnvState [i], (EnvInput [i])[1]))[v], v= 1, 2 are, correspondingly, first and second components of the resulting vector for the source operation op.

Only extended operations will be considered further. To simplify the notation, op will be used instead of $op^{(ext)}$ meaning the mapping: EnvStateSet × EnvInputSet → EnvStateSet × EnvInputSet.

A binary meta-operation named *concatenation* and denoted by * is defined over extended object operations of a given environment.

Let op_i be an extended operation of an object o_i and op_j be an extended operation of an object o_j. Then op_i*op_j is an operation meaning the mapping {EnvState} × {EnvInput} → {EnvState} × {EnvInput} in the following way:

op_i*op_j (EnvState, EnvInput) = op_j (op_i (EnvState, EnvInput)).

Let Op = {op_1, …, op_k} be a set of all extended operations for all objects of a given environment including also an empty operation which provides a trivial same-to-same mapping {EnvState} × {EnvInput} into {EnvState} × {EnvInput}.

Then, using the introduced meta-operation of concatenation a semi-group of all words in the Op alphabet can be considered. It will be denoted further by F_{Op}.

Let G be a semi-group with a unit, generated by the set of generators Y = {y_1,…, y_k}. G is called free partially commutative semi-group, if it is defined by a finite set of determining assumptions of type $y_i y_j = y_j y_i$ [4].

Two objects o_i and o_j are named *commutative* if and only if for any extended operation op_i of the object o_i and for any extended operation op_j of the object o_j ∀EnvState ∀EnvInput

op_i*op_j (EnvState, EnvInput) = op_j*op_i (EnvState, EnvInput).

A notion of process is introduced to provide a possibility of a programmed control over an environment. It is started by introducing the following primitives:

TimeConstraint = {$t_0 + n * \Delta t \mid n \in \{0, 1, 2, \ldots\}$}, where t_0 and Δt are non-negative integers

BasicPred = {$\pi_1, \ldots, \pi_r$}, where π_j is a predicate symbol of arity n_j

Condition = $\pi_j(o_{i_1}.\text{State}, \ldots, o_{i_{n_j}}.\text{State})$, where $\pi_j \in$ BasicPred

Assertion = (Condition, TimeConstraint)

Situation = (Object, Assertion)

SituationBatch={Situation}

Let $\Theta = \{o_1, \ldots, o_n\}$ be an environment.

A *process* over an environment Θ is a tuple $P = \langle \Theta, \Sigma, B, \Gamma, b_s \rangle$, where:

- Σ – is an ordered finite set of situations, such that $\forall \sigma \in \Sigma$ $\sigma.Object \in \Theta$
- B – is a set of situation-batches, such that $\forall b \in B$ $b \subseteq \Sigma$, $b_e = \varnothing \in B$ is the end situation batch
- Γ – is a relation $\{(\sigma, b, \omega) \mid \sigma \in \Sigma\}$, where $b \in B$, $\omega \in \sigma.Object.Operatons$
- b_s – is the start situation batch

An interpretation is built for a given process, if the initial state of the environment and following functions are defined:

- initial EnvState: $es_0 = (st_0^{(1)}, \ldots, st_0^{(n)})$
- a function f_{op}: {EnvState}×{EnvInput}→{EnvState}×{EnvInput} for each operation symbol op ∈ Op
- a function $\rho_{\pi_j} : o_{i_1}.\text{StateSet} \times \ldots \times o_{i_{n_j}}.\text{StateSet} \to$ {TRUE, FALSE} (n_j is the arity of the corresponding predicate symbol) for each predicate symbol.

The semantics of a process interpretation is described below via the following execution algorithm.

To ensure consistent execution of the process, concurrent execution of two and more operations of the same object is not allowed. So, in case if there are two pending situations, ready to be executed at the same time, it must be chosen which one to execute first. The other situation should stay in a pending state. To implement this, the set of situations (Σ) is defined above as ordered, as well as corresponding checks are performed in the execution algorithm.

The data structures and functions used in the execution algorithm are listed below.

- **InputScene** – the initial scene (input of the algorithm)
- **Pending** – a set of situations that are waiting to be executed
- **Running** – a set of situations Σ′, for which ∀σ∈Σ′, operation(σ) is currently running
- **CurrentEnvInput** – the current environment input
- **nextNode(σ)** = b, if (σ, b, ω) ∈ Γ
- **operation(σ)** = ω, if (σ, b, ω) ∈ Γ
- **receivedMsg(σ)** = TRUE, if a message needed for operation(**σ**) is in **CurrentEnvInput (in the message queue of the corresponding object)**.
- **canExecute(σ)** = **receivedMsg(σ)** & σ.Assertion.Condition, where **σ** is a situation.
- **beginExecution(σ)** – removes the message needed for operation(**σ**) from **CurrentEnvInput (from the message queue of the corresponding object)**.
- **finalizeExecution(σ)** – changes the state of the corresponding object and adds the ObjectReaction of operation(**σ**) to **CurrentEnvInput (CurrentEnvInput = CurrentEnvInput** ♀ ObjectReaction**)**.
- **executed(σ)** = TRUE, if duration(operation(**σ**)) time has passed from the corresponding **beginExecution(σ)**.
- **add(dest, sitBatch)** – adds all situations of the situation batch **sitBatch** to the set **dest** (no duplications).
- **remove(source, sit)** – removes situation **sit** from the set **source**.
- **move(sit, source, dest)** – removes situation **sit** from the set **source** and adds to the set **dest** (no duplications).
- **foreach(σ, cond(σ))** – iterates sequentially over any situation **σ** for which a given condition over **σ - cond(σ)** is true, according to the order defined in Σ.

Execution Algorithm

```
Pending = Running = ∅;
CurrentEnvInput = InputScene.EnvInput;
add(Pending, bs);
for(t = InputScene.t; ; ++t)
begin
  foreach(σ, σ∈Running & executed(σ))
  begin
    finalizeExecution(σ);
    remove(Running, σ);
    add(Pending, nextNode(σ));
  end

  foreach(σ, σ∈Pending & canExecute(σ)
   & t∈σ.Assertion.TimeConstraint
   & (∀σ1∈Running σ1.object≠σ.object))
  begin
    move(σ, Pending, Running);
    beginExecution(σ);
  end

  if(Running = ∅
       & (∀σ∈Pending ¬canExecute(σ)))
    exit;
end
```

The **execution is successful** if **Pending** = ∅ at the end, otherwise the **execution is failed**. For a successful execution, the result of the algorithm is the last scene before the end of the algorithm (output scene). The tuple (Pending, Running,

CurrentTime, CurrentEnvInput) will be called an *execution state* of the algorithm.

Two processes defined over the same environment will be called **functionally equivalent** if and only if for every input scene the execution of both processes either fails or the environment inputs and environment states in output scenes are equal for all interpretations.

We will denote the equivalence of P_1 and P_2 by $P_1 \sim P_2$.

The functional equivalence problem for the case when in a given environment there are no commutative objects was considered in [2]. It was shown, that this problem could be reduced to the equivalence problem of multidimensional multitape automata, which is solvable [5]. It can be shown, that the result also holds when the extensions introduced here are considered.

The equivalence problem of processes in environments with commutative objects is unsolvable in general. The case when there is only one object having 2 or more operations will be considered below.

An equivalent representation of a process (named *sequential execution scheme of the process* – SESP) will be constructed next. This representation is more convenient for further considerations.

Let $P = \langle \Theta, \Sigma, B, \Gamma, b_s \rangle$ be a process and $\Sigma=\{\sigma_1, \ldots, \sigma_\eta\}$. Let also $T_i = \{t_{0i} + n * \Delta t_i \mid n\in\{0, 1, 2, \ldots\}\}$ is a time constraint for a given situation σ_i of Σ. Let Δt be the least common multiple for all Δt_i, t_0 be the maximum of all t_{0i}.

The set of states of the SESP(P) corresponds to the set of execution states of the execution algorithm (taking into account the periodicity of time constraints). It is defined as $2^\Sigma\times R\times T\times\{EnvInput\}$, where 2^Σ is the set of pending situations, $R = \{[(\sigma_1, \tau_1), \ldots, (\sigma_b, \tau_b)] \mid \sigma_i.object \neq \sigma_j.object, 0<\tau_i<duration(operation(\sigma_i))\}$ is the set of running situations (τ_i is the time remaining for the completion of operation(σ_i)), $T = \{0, \ldots, t_0+\Delta t-1\}$ is the set of possible values of current time, {EnvInput} is the set of current environment inputs. Let $NextT(t) = t+1$ if $t < t_0+\Delta t-1$, $NextT(t) = t_0$ if $t = t_0+\Delta t-1$. A transition is defined from state $s^{(1)} = (P^{(1)}, R^{(1)}, t^{(1)}, IE^{(1)})$ to state $s^{(2)} = (P^{(2)}, R^{(2)}, t^{(2)}, IE^{(2)})$, where $P^{(1)}, P^{(2)}\in 2^\Sigma$, $R^{(1)}, R^{(2)}\in R$, $t^{(1)}, t^{(2)}\in T$, $IE^{(1)}, IE^{(2)}\in\{EnvInput\}$, if $t^{(2)} = NextT(t^{(1)})$ and $s^{(2)}$ can be reached from $s^{(1)}$ by one step of the execution algorithm for $t = t^{(1)}$ for some interpretation.

The execution algorithm is modified to work with SESP in the following way. As the sets *Pending*, *Running* and *CurrentEnvInput* are already encoded in the states of SESP, we just need to start from the state corresponding to the input scene and go to a next state corresponding to the *Pending*, *Running* and *CurrentEnvInput* sets of the original algorithm, executing the operations and changing the states of objects as in the original algorithm.

Let $StateT(t) = t$ if $t < t_0$, $StateT(t) = t_0 + (t-t_0 \bmod \Delta t)$ otherwise. Let $InputState(S_I) = (b_s, \varnothing, StateT(S_I.t), S_I.EnvInput)$ be the state corresponding to the input scene S_I, $OutputState(S_O) = (\varnothing, \varnothing, StateT(S_O.t), S_O.EnvInput)$ be the state corresponding to the output scene S_O.

The following lemma states the correspondence between a given process and its SESP.

Lemma 1. For every interpretation I and every input scene S_I:

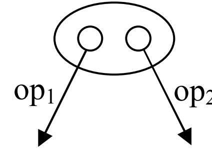
a) if the execution of the process completes successfully with an output scene S_O, there is a path in the SESP from IS = InputState(S_I) to OS = OutputState(S_O), and the execution of the SESP with the input scene S_I reaches OS and vice versa;

b) if the execution of the process fails, the execution of the SESP never completes and vice versa.

The set of all operations of all objects of an environment Θ will be denoted by $\Theta_{op} = \bigcup_{i=1}^{n}\bigcup_{j=1}^{m_i} o_i.op_j$. Let Y be a partially commutative alphabet [3], $Y = \bigcup_{i=1}^{n}\bigcup_{j=1}^{m_i} \{ op_{ij}^{1}\ldots op_{ij}^{duration(o_i.op_j)} \}$ where $op_{ij}^{(1)}\ldots op_{ij}^{(duration(o_i.op_j))}$ is a representation of a given operation j of object i as a sequence of sub-operations that are executed within one time unit of the operation duration. Evidently, if $o_i^{(1)}.op_j^{(1)} * o_i^{(2)}.op_j^{(2)} = o_i^{(2)}.op_j^{(2)} * o_i^{(1)}.op_j^{(1)}$ then $op_{i_1 j_1}^{(k)} op_{i_1 j_1}^{(l)} = op_{i_1 j_1}^{(l)} op_{i_1 j_1}^{(k)}$.

It is easy to transform a given SESP into a regular expression over the alphabet Y – for each node one has just to add node transitions for missed operations. For a given SESP P the corresponding regular expression will be denoted R(P). The figures below demonstrate the transformation from the source process to automaton that recognizes corresponding regular expression.

Fragment of a source process P

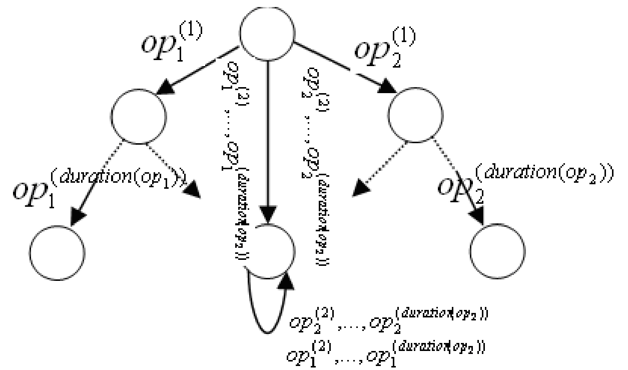


Sequences of suboperations for op_1 and op_2

$$op_1 = op_1^{(1)}\ldots op_1^{(duration(op_1))}$$

$$op_2 = op_2^{(1)}\ldots op_2^{(duration(op_2))}$$

Fragment of the automaton corresponding to regular expression R(P)



Let X be a finite partially commutative alphabet, $X_1, \ldots X_n$ be the decomposition of the alphabet X on disjoint non-commutative subsets by a relation ξ. Let R_1 and R_2 be regular expressions in the alphabet X. If for every word $w\in R_1$ there exists a word $w'\in R_2$, that coincides with w up to commutation of symbols from different subsets of the alphabet X and, vice versa, if for every word $w'\in R_2$ there exists a word $w \in R_1$, that coincides with w up to commutation of symbols from different subsets of the alphabet X then regular expressions R_1 and R_2 will be called ξ -equivalent and denoted by $R_1 \sim R_2(\xi)$ [3].

Lemma 2. For any SESP P_1 and P_2 in the environment Θ with commutative objects $P_1 \sim P_2 \Leftrightarrow R_1(P_1) \sim R_2(P_2)(\xi)$.

Theorem 1. If there exist two commutative objects in an environment Θ which has more than two operations then the equivalence problem of processes in the environment is unsolvable.

This can be proved using the technique similar to [6], i.e. the problem is reduced to the equivalence problem of non-deterministic multitape automata.

Theorem 2. The problem of functional equivalence of processes in an environment with commutative objects, when there is only one object which has more than one operation, is solvable.

The proof implies from the result obtained in [3].

REFERENCES

[1]. P. Raulefs, "The Virtual Factory", IFIP World Computer Congress '94, v. 2, pp. 18-30, 1994.

[2]. H. A. Grigoryan, "Equivalence of Processes in an Object-Oriented Environment", Reports of the National Academy of Sciences of Armenia, vol. 108 (1), pp. 50-59, 2008.

[3]. А. С. Шукурян, "Эквивалентность регулярных выражений над частично коммутативным алфавитом", Кибернетика и Системный Анализ, №3, стр. 3-11, 2009.

[4]. А. Б. Годлевский, А. А. Летичевский, С. К. Шукурян, "О сводимости проблемы функциональной эквивалентности схем программ над невырожденным базисом ранга единица к эквивалентности автоматов с многомерными лентами", Кибернетика, №6, стр. 1-7, 1980.

[5]. H. Grigorian, S. Shoukourian, "The Equivalence Problem of Multidimensional Multitape Automata", Journal of Computer and System Sciences Volume 74, Issue 7, November 2008, pp. 1131-1138.

[6]. В. А. Тузов, "Проблемы разрешения для граф-схем перестановочными операторами. II", Кибернетика, №5, стр. 28-32, 1971.