IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 21, 2020, accepted November 5, 2020, date of publication November 16, 2020, date of current version November 23, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3037798

Mone: Mutation Oriented Norm Evolution

XIANCHANG WANG[1,2,3], RONGHAO FU[1,2], AND RUI ZHANG[1,2]
[1]College of Computer Science and Technology, Jilin University, Changchun 130012, China
[2]Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China
[3]Chengdu Kestrel Artificial Intelligence Institute, Chengdu 610000, China

Corresponding author: Rui Zhang (rui@jlu.edu.cn)

This work was supported in part by the Natural Science Foundation of Jilin Province under Grant 20190201273JC.

ABSTRACT A norm regulates the run-time behavior of the agent with the action and condition to trigger the action. Because of the incomplete understanding of the world, the *result* of the action may be different for the same agent with the 'same' context, or unintended different context. To study this phenomenon, the classical norm definition is extended from the condition-action pair to cover the *expectation*, in order to verify the result of the action. The Norm evolution can be defined as a gradual process which makes a norm more complete and effective. In the terminology of evolution, a norm is called *mutated* if the result contradicts to the expectation, i.e. at least one of the expected conditions is invalid. At run-time, norms are executed in series. A mutation brings new knowledge to the following states and might affect the later execution of the norms. Such knowledge provides will help the norm designer to complete the definitions. A mutation based norm evolution (Mone) method is proposed in this paper to detect the mutations, to propagate the evidence and to crossover the norms for completeness. The method is formalized in the Description Logic, and implemented with the algorithms for mutation detection and norm crossover. The case study illustrates the Description Logic $\mathcal{ALCI}$ of the method and shows the potential to evolve the norms autonomously in the Blackboard system, GBBopen.

INDEX TERMS Mutation, Norm Revision, Blackboard-system, Multi-Agent System.

I. INTRODUCTION

One of the most famous multi-agent system frameworks is the blackboard based system (BBS) whose history goes back to 1980's [1]. Open source frameworks such as GBBopen,[1] Repast [2] and Repast HPC [3] are used as the backbones of advanced science, engineering, and policy analysis. The BBS architecture provides a flexible framework to solve complex problems with *blackboard, knowledge source (KS)* and *control shell*. The blackboard serves as a space for common knowledge sharing from various KSs. Each KS can be regarded as an agent that operates autonomously and changes the shared knowledge on the blackboard in a controlled manner with predefined functions and trigger events, which are the *action* and the *conditions* of the norm.

The flexibility of BBS leads to complex interactions between the KSs and blackboard, through the transition of the states by the execution of the predefined norms (or the norm's actions). Norms are designed to be executed in series such that the result of a norm triggers the next norm. But it is not always the case as the context can never be studied completely. Some executions may follow the expectation, some may lead to exceptions. However, there may be a number of exceptional results that are identified as significant for other norms. Such an exceptional case is called a mutation i.e. the action is successfully triggered but the result of the action contradicts the expected outcome (negation of one of the expected outcome conditions). Mutations often, if not always, emerge in the real world applications because the predefined norms are always an incomplete description of the regulatory knowledge.

To reconcile this incompleteness of predefined knowledge, the norms should and have to evolve. Then the questions arise as what is the new knowledge for the evolution of the norms? where does the new knowledge come from? and how does the norm evolve with such knowledge?

To answer these questions, the BBS system is abstracted into an agent-based scenario. The norms are extended to incorporate expected results as expectation, in addition to the action and the trigger conditions that fire the action. The KSs are abstracted as norm executers that verify the knowledge on the blackboard whether its norm is triggered and execute the action and publish the result to the blackboard. The

The associate editor coordinating the review of this manuscript and approving it for publication was Hiram Ponce.

[1]http://gbbopen.org/index.html

blackboard is abstracted as a state transactor that a state is denoted by all the knowledge on the blackboard and transits to another state, denoted with all the knowledge provided by all the KSs that executed the triggered norms. The system operates with a series of actions, i.e. to execute several norms sequentially in the transitions of states.

For example a robot that sweeps the room should follow a norm like the following:

- Trigger: Time≥8:00AM
- Action: Sweep
- Expectation: Finish sweeping and Time≤8:30AM

The norm is executed autonomously if its trigger conditions are satisfied. It should finish sweeping as the expectation prescribed, but not necessarily as the norm might mutate because of power shortage, broken arms, obstacles, etc. which can never be exhausted in an open world. The robot may follow the same series of norms everyday according to the designed norms. Such kind of routine executions do not provide any 'new' knowledge to the designer therefore it is hard to revise the norms in such routines. On the other side, mutations may take place in some of the days, the unexpected result may provide extra knowledge to the execution series and affect the execution of later norms.

This paper explores the evolutionary behaviors of norms, i.e. inheritance, mutation and crossover. The result of the executed action can be either coherent or contradictory to the expectation of the norm. The norm is called *mutate*d if any of the expectation is violated by the result. In the different series of norm executions, if a norm, say x mutates on one track while performs normally on another, x is supposed to evolve with some extra knowledge. We use the Layered Execution Graph (LEG) defined in the modeling section to analyze the different norm executions. It is found that other than mutated, in the track that x does not mutate, there exist other norms, say $y, z \ldots$ which mutate ahead of x. The mutations of $y, z \ldots$ provide extra knowledge not prescribed in the norm construction stage. It is also found that such extra knowledge can be *inherit*ed to x through the tracks of the LEG. x can evolve by *crossover* with the mutated norms $y, z \ldots$ The paper proposes a novel mutation oriented norm evolution method: Mone. It enriches the trigger condition of the norm that mutates, with the expectations of some other mutated norms. The contributions of the paper lay in 3 aspects:

- To find the relationship between different executions of the same norm, from a mutated one to a well-executed.
- To prove the existence and inheritance of the extra knowledge from mutations.
- To provide algorithms to construct the Layered Execution Graph, to detect the mutations and to evolve the norms.

II. STATE-OF-THE-ART

Knowledge based systems are always facing the incompleteness of the knowledge. This is especially challenging in a multi-agent blackboard based system [1] where common knowledge can be changed by different knowledge sources [4]. One of the most influential aspects leads to the 'abnormal' executions of the norms, which regulate the runtime behaviors of the agents. It demands the norm evolution with the growth of knowledge.

Current norm evolution studies mostly fell into 3 categories, norm formalisation, norm conflict resolution and norm failure localisation. Fisher *et al.* discussed the relationship between computational logic and agents [5] in agent specification, implementation and analysis. Many logical frameworks are proposed to formalize the domain [6]–[8]. Besides, model checking [9] and belief-desire-intention framework [10] are also applied for norm representation. But this is not in our focus because we emphasize on the use of mutation rather than its representation.

Vasconcelos *et al.* proposed the norm with constraints and a framework to detect and resolute conflicts in [11]. Silvestre and Da Silva developed a method to resolute conflicts among multiple norms [12]. Santos *et al.* classified [13] the conflict resolution approaches into direct and indirect detections and surveyed the progresses. Kayal *et al.* proposed to resolve conflicts in social commitment automatically through a conflict resolution model based on relevant user values such as privacy and safety [14]. This thread focuses on the resolution of a given norm set, which is not the problem we want to solve.

Passos *et al.* found the error-prone nature of multi-agent systems and focused on the behavior failures via spectrum-based fault localisation [15]. It has been improved by the accuracy graph method in 2017 [16]. Alechina *et al.* discussed the imperfect monitor of runtime norm executions [6]. Huang and Alexander explored the semantic mutation testing to assess tests and program robustness [17]. The Silk framework was proposed to monitor and resolve the conflicts of norms [18]. The work in this thread focuses on the conflicts between different norms. The common solution is to evaluate and place a priority on the conflicting norms. It is different from the Mone method in that we focus on the different behaviors of the same norm on multiple executions and explore the possible reasons inherited from the execution results of other norms that are mutated and generate expected knowledge useful for the evolution.

III. MOTIVATING EXAMPLE

In an agent-based system, a sweeping robot cleans the room. The robot automatically starts cleaning from *8:00 AM* if set in the *Auto* mode. It is expected to take the robot *less than 30 minutes* to complete the job. Everyday, the robot finishes sweeping before 8:30 AM; except one day, the robot was stuck by an obstacle and did not manage to remove it, which resulted in the failure of the task. To detect the reason, the robot was set to the *monitor* mode, and tested for exceptions. In this case, an obstacle was found and its weight was beyond the ability of the robot, therefore removed manually by the monitor. Then without the obstacle, the job was finished in time. Yet another day, the job failed again.

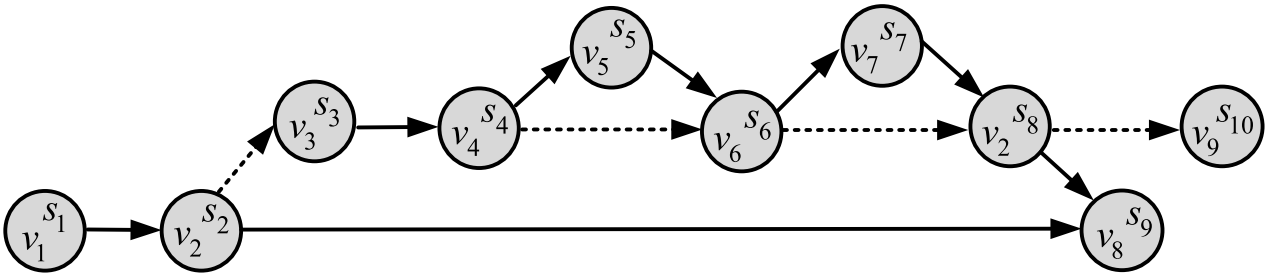


FIGURE 1. Execution graph of the motivating example. The dashed arrows represent mutations, the continuous arrows represent the normal execution (v_i represents the norm, s_i represents the state, v_1: task starts; v_2: sweep room; v_3: monitor starts; v_4: obstacle tests; v_5: obstacle removal; v_6: power tests; v_7: power charges; v_8: task completes; v_9: task fails).

In another run of the monitor mode, it was found that the robot ran out of battery half way of the job. By manually charging, the robot recovered and finished the job.

With the above findings, it's rational to refine the norms that regulate the robot with the evidences found in the monitor mode, such that the conditions to start the sweeping job should incorporate not only *Time*==*8AM* and *Mode*==*Auto*, but also *Obstacle*≤*1kg* and *Battery*≥*80%*.

To study the different behaviors of the agent in multiple series of norm executions, we introduce the so-called layered execution graph. To describe the execution path, let v_i represent the name of *norm*, s_i represent the *state*. Figure 1 gives an example of the graph.

The graph represents multiple series of executed norms. In the first state s_1, it starts from a root vertex as the common *head* of the different execution series. Each state denotes a set of conditions, such as *Time*==*8AM* and *Mode*==*Auto*. States grow in a horizontal direction sequentially. Each vertex denotes the (multiple) execution of a norm, triggered in the conditions of the state. Directed edges represent the order of the executions. The multiple outgoing edges from a vertex denote the different execution results of the norm. The solid edge denotes that the norm has been executed as expected; the dashed one denotes the result has some condition that is not prescribed by the norm. There are some execution paths in Figure 1. One series of execution is (v_2, v_8) that the task starts and completes normally, while another is (v_2, v_9), which indicates the robot is stuck and the task fails. Yet another series is $(v_1, v_2, v_3, v_4, v_5, v_6, v_2, v_8)$ which indicates that the obstacle detected by the test is removed and the sweeping can be completed. Yet another series is $(v_1, v_2, v_3, v_4, v_6, v_7, v_2, v_8)$, which indicates the robot is start sweeping in the monitor mode and the power shortage problem is settled by recharging, after the power test which is in turn an internal system test.

IV. MODEL

For better formalize the scenario, the Description Logic $\mathcal{ALCI}$ (attributive concept description language) is used with a real world interpretation $\mathcal{I}$ of the TBox (Terminological Box) and ABox (Assertional Box). The Description Logic is divided into a TBox part and an ABox part. The

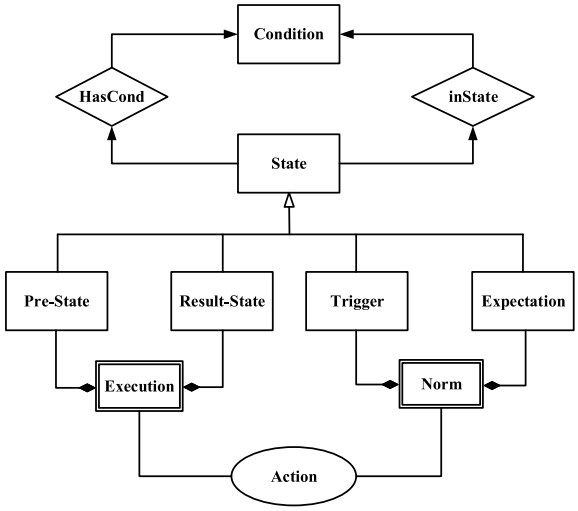


FIGURE 2. ER Diagram of Mone.

TBox defines the structure of the domain and includes some axioms, the ABox describes examples of the domain and including some axioms asserting.

The major entities such as *norm, condition, state, action* are defined as DL classes and the auxiliary entities such as *time, location, weight,* etc. are left as predefined domain attributes for data properties. In a general agent-based context, a *state* describes the environment with the features as *conditions*; a *norm* regulates the way an *agent* functions given a certain state. Traditional norms describe the regulation in a pair (*request*, *action*), where *request* ⊆ *state* describes the necessary conditions to trigger the *action*.

But it is not good enough to study the outcome of the action in such a definition. In our findings, a norm should clarify two states:

- The state satisfies the norm's *request* in order to trigger the action;
- The state of the agent after the action, as the expected result prescribed in the norm's *expectation*.

Figure 2 gives the conceptual model of the Mone method. The *Condition* and *State* are first class entities, with *Trigger, Effect* and *Expectation* as the sub-class of state. A state may have multiple conditions, while a condition can be *inState* of

multiple states. An *Action* connects the *Trigger* state to the *Effect*. A *Norm* regulates the agent that given the trigger state, the execution result of the action should be the *Expectation*.

If the outcome of the norm execution is not as expected, a exceptional situation occurs. From a norm definition point of view, it is called a *mutation* that the prescription of the norm is not complete and effective. To clarify the scenario, we have the following definitions and theorems.

Definition 1 (Condition): A condition describes the state in one dimension/perspective. It is formalized as an individual c of the prime class Condition with the ABox assertion as

$$Condition(c)$$

In certain context, a condition can be either satisfied or unsatisfied. Therefore, we define the class *Satisfied* as the subclass of *Condition* for those interpreted *true* in the context, and $Unsatisfied \equiv Condition \sqcap \neg Satisfied$. Then if a condition c w.l.o.g. is satisfied, there exists a pair-wise counter condition *not c* that is unsatisfied in the same context. Here in the syntax of DL, negation is not applicable on an individual, therefore $\tilde{c}$ is used for the counter condition of c.

The condition class can be further classified into subclasses as *time-related* condition, *space-related* condition and other facilitated set of domain individuals.

For example, there are two individual conditions p_1, p_2 where p_1 indicates that *the mode of the robot is set as auto* and p_2 indicates *the current time is NOT 8 AM*. If the current context interprets that *the robot is in auto mode and the time is 8 AM*, then the following ABox assertions hold: $Satisfied(p_1)$, $Satisfied(\tilde{p_2})$.

Definition 2 (State): A state describes the features of the context, as a set of conditions, formalized as a DL class

$$State \equiv \exists hasCond.Condition$$

where hasCond is the object property from the class State to Condition.

A state s is called the *sub-state* of another state s' iff the conditions in s are also in s', i.e.

$$subState(s, s') \leftrightarrow inState : s \sqsubseteq inState : s'$$

where $inState \equiv hasCond^{-}$, and $a : b$ is the syntax sugar of $\exists a.\{b\}$ for simplicity.

A state s is called *before* another state s' iff there exists strictly sequential time related conditions between the states.

$$before(s, s') \leftrightarrow hasCond(s, t), hasCond(s', t'), t < t'$$

where $<$ is the strict order in the time domain.

Suppose that p_3 stands for the condition that the robot finishes sweeping in half an hour. The condition p_4 stands for the robot changes its mode into *Finish*. If one day at 8:30 AM, it is observed that the sweeping task has not finished, then the state is modeled as the ABox assertion $\neg hasCond(s_1, p_3)$.

To describe the behavior of the agent, the prime concept *Action* is defined.

Definition 3 (Action): An action describes the behavior of the agent as the mapping from the Trigger state to the Result state. Formally, it is a functional object property

$$act \sqsubseteq Action$$

that projects a state to, if not the same, another state.

To detect the abnormal behaviors of the norm system, we extend the classical norm definition to incorporate the expected outcome of the execution.

*Definition 4 (Norm): A norm is a named pair act(tri, exp), that denotes the **act**ion, the **tri**gger conditions and the **exp**ected resulting conditions after the execution of the action. Formally, a norm is an ABox assertion*

$$act(tri, exp)$$

where act $\sqsubseteq$ Action, State(tri) and State(exp) holds.

A norm n defined as $act(tri, exp)$ is called *applicable* in the state s iff $subState(tri, s)$ holds; the action act is called *executable* in s, iff n is applicable in s.

*Definition 5 (Execution): The execution e of a norm act(tri, exp) is a named pair act(pre, res), that denotes the executed **act**ion, the **pre**state that triggers the execution and the **res** state after the execution. Formally, e is an ABox assertion*

$$act(pre, res)$$

where subState(tri, pre) and State(res) holds.

In most of the cases the agent executes the norm and the result state is coherent with the expectation of the norm. But this is not always the case because in the real world, a norm is hardly perfect defined for the partial understanding of the constantly changing context. The execution is called *exceptional* if its *res* is not coherent with the *exp* of the norm. Such exceptions provide extra information not expected in the definition of the norm system. In the exceptional execution, the norm is called *mutated* because it behaves differently from expectation.

Definition 6 (Mutation): A norm n=act(tri, exp) is in mutation iff there exists an execution e=act(pre, res) such that there exists at least one condition in res which is conflict from a condition in exp. Formally Mutation(n,e) iff the following class is satisfiable (has individual in any interpretation).

$$inState : res \sqcap \neg inState : exp$$

Here the complex class is satisfiable in DL semantics iff in any interpretation $\mathcal{I}$ there exists an individual c such that $c^{\mathcal{I}} \in (inState : res \sqcap \neg inState : exp)^{\mathcal{I}}$, which says there exists a condition that is the result of the norm execution, but not expected in the norm. According to the Open-world Assumption of the DL semantics, negation as failure is not assumed. The satisfiability of the class is validated with all the named entities only.

The motivating example is shown in Figure 1. A norm $n_1 = act_1(tri_1, exp_1)$ is applicable in any state s_1 such that $subState(tri_1, s_1)$, where exp_1 is the expectation with $hasCond(exp_1, p_3)$ and $hasCond(exp_1, p_4)$. If the state s_2 is

observed to have the condition $\neg p_3$ after the a_1 executes the norm r, then r is mutated in s_2 as $\neg hasCond(s_2, p_3)$ contradicts the expectation e_1. With a general purpose DL reasoner, it is easy to detect which norm has been mutated at runtime.

A mutated norm provides extra evidences not expected in the design phase of the norms. Such evidences may affect the execution of the norms afterwards. Therefore, it is rational to track the path of the norm executions.

Definition 7 (Path): A path describes a sequence of norm executions. It is formalized as a list $\bar{e}$, with element $e_k = act_k(pre_k, res_k)$ that for each k, $1 \leq k < |\bar{e}|$, the following TBox holds

$$inState : res_k \sqsubseteq inState : tri_{k+1}$$

where tri_{k+1} is the trigger state of the norm e_{k+1} executes. In path, the norms are executed in a series of states that triggers the corresponding norms. For a certain state, it is not restricted that only one norm can be executed, i.e. the state will trigger all the applicable norms. Given a sequence of state transitions, if each state triggers only one norm, and the execution results are not conflict to the following state, then a path is constructed; if in some state, multiple norms are triggered, the path will split to multiple successors e^l_{k+1} while $1 \leq l \leq N_{tri}$, where N_{tri} denotes the number of norm that triggered by some conditions. As norms are executed in the time domain, a path will grow into a tree and even an acyclic directed graph.

The agent should complete the task according to the designed norms. However, although the paths are predicted to be the same, mutations do exist because of the incomplete understanding of the constantly changing world. A mutation may terminate the path, while it may trigger other norms and continue the path as well. No matter in which position the norm is mutated, it marks the unexpected behavior in the trigger context, and indicates the necessity of evolution for the mutated norm. Therefore, the mutations in the paths should be the anchors for the norm evolution.

In the real world, agents are usually designed to complete a predefined task, repeatedly if necessary. For example the sweeping robot will sweep the floor everyday. The paths will be high-probably the same if the robot works normally. To find the mutation (i.e. to position which norm to evolve), it is rational to take as a whole such 'repeated' paths. By merging all the paths for the same agent on the same state transitions, a directed acyclic graph is constructed.

Definition 8 (Layered Execution Graph, LEG): A LEG is a triple $G = (V, D, L)$ that records multiple paths in the same series of state transitions, in which the vertex $v \in V$ denotes the norm execution; the directed edge $d = (e_i, e_j) \in D$ denotes the order of executions in the recorded path; the layer $l \in L$ denotes the state and the time domain in which the norm execution takes place.

LEG is constructed to position the mutation of norm execution, therefore it records the paths of the same agent in the 'same' state transitions. To differentiate the mutated execution from the normal ones, the former connects to the next vertex with a dashed edge in contrast to the common solid edges.

The agent can execute a series of norms multiple times in the 'same' context, such as in the motivating example, the robot sweeps the floor everyday. But the context may vary in an unexpected fashion that the action may have different result from the 'same' trigger state. In the real world semantics, the result state of the action may either obey the norm or mutate it. For example, the state of layer s_8 in Figure 1 incorporates the trigger state of the norm $a_2(hasCond : p_2, hasCond : p_3)$. But the execution result is $a_2(hasCond : p_2, \neg hasCond : p_3)$ that the agent did not finish the sweeping task by 8:30 AM.

Theorem 1 (Condition Inheritance): A condition in a state will be inherited to the state after, if not explicitly changed. Formally, given a condition c w.l.o.g. that

$$before(s, s'), hasCond(s, c) \rightarrow \begin{cases} hasCond(s', \tilde{c}) & if \quad Satisfied \sqcap inState : s'(\tilde{c}), \\ hasCond(s', c) & else \end{cases}$$

Recall the example that the condition $p_1 : mode == auto$ and $p_2 : Time == 8AM$ in Figure 1. The state s_1 satisfies that $hasCond(s_1, p_1)$ and $hasCond(s_1, p_2)$. Afterwards, the state s_2 still satisfies $hasCond(s_1, p_1)$, as there is no explicit change of the mode of the robot. But p_2 is changed for the time elapses then $hasCond(s_2, p_2)$ is not valid anymore. Theorem 1 shows that in a track of a LEG, the afterwards state tends to inherit the conditions if not explicit objected. In the Open-World Assumption, such resistance is of great value for inference with incomplete knowledge.

Theorem 2 (Evidence Existence): A norm is triggered in a state, and mutated in the state after, then there exists some condition in the former state which is not the trigger condition of the norm, that has been inherited to the state after and contradicts to the expectation of the norm. Formally, given $before(S, S')$, a norm $R = A(T, E)$ and its execution $A(T, S')$, there exists a condition C that

$$\neg inState : E \sqsubseteq inState : S' \leftrightarrow$$
$$\{C\} \sqsubseteq \neg inState : T \sqcap inState : S, \neg hasCond(E, C)$$

Proof: Assume that $\neg inState : T \sqcap inState : S \sqsubseteq \bot$, then for each condition C, $hasCond(S, C) \rightarrow hasCond(T, C)$; S triggers $R \Leftrightarrow subState(T, S) \Leftrightarrow$ for each C, $hasCond(T, C) \rightarrow hasCond(S, C)$; Then $T \equiv S$. As the action A is functional, $S' \equiv E$, which contradicts to the assumption.

Theorem 2 assumes the actions are executed functionally, i.e., for the same state, the action result should be the same. Although an action may be not functional in the real world, non-functional actions are not in the scope of this paper. Besides, there exist unexpected outer forces in the real world to change the conditions, which are not considered in our

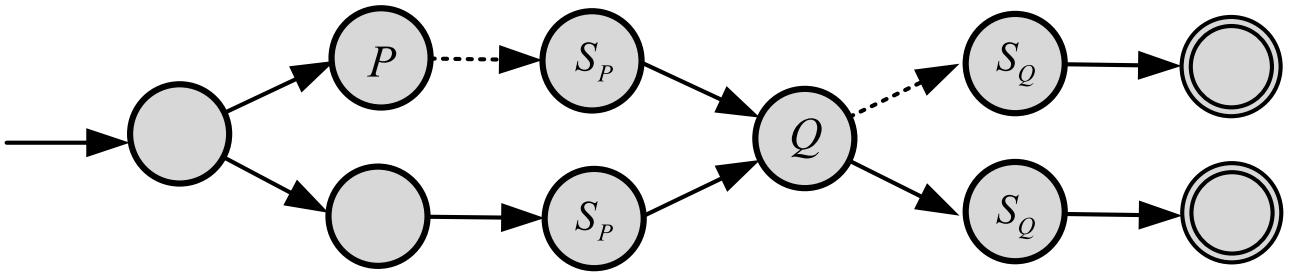


FIGURE 3. Norm mutation resistance, the dashed arrows represent mutations, while the continuous arrows represent the normal execution.

scenario either. Then mutations are the main cause of the norm evolution. □

Theorem 3 (Pairwise Mutation): Given two tracks of norm execution sequences of the same agent, a norm is obeyed in one but mutated in another, there should be some extra evidence lead by another mutation ahead in the obeyed track, that is inherited to but not specified in the trigger state of the norm. Formally, given two tracks α and β that w.o.l.g. a norm $R = (T, E)$ is mutated in α and obeyed in β. There exists some condition C as the result of the execution of the norm $R' = (T', E')$ in β ahead of R, such that $\neg hasCond(E', C)$ mutates R', and C is inherited to the norm R that the state $\exists hasCond.(\{C\} \sqcup \exists inState.T)$ triggers R and prevents R from mutation.

Proof: It is assumed that no outer force is detected to change the conditions in the states on the tracks, therefore the conditions may change only because of the action execution. All the actions are functional, then the norms on each vertex are either obeyed or mutated. As the norm R is applicable on both α and β, there exits some evidence inherited to the norm R through α that leads to the different performance from R on β. If no norm is mutated on α ahead of R, the inherited conditions should be the same as those on β, then the action on both tracks will produce the same result. This is against the fact that R is obeyed, so there exists some norm mutated on α before R. □

In Figure 3, the left path illustrates α and the right pictures β. The dashed arrows illustrate the mutations of the norms. From Theorem 2, the reason R is mutated on Q' but obeyed on Q is because there exists another norm mutated on R' and the conditions are inherited to Q by Theorem 3.

With the open world assumption, there is always some condition(s) not considered at the time of norm construction. Moreover, there might exist some unnecessary conditions that were added to the trigger of the norm not on purpose. Intuitively, the expectation of the norm is regarded as the precise result from the norm action.

V. ALGORITHMS

Norms execution process based on BBS is mainly implemented by Algorithm 1.

The Algorithm 1 shows how the norms execute on the blackboard system. The *Control Shell* (Line 1 and Line 16) is one of the three major components of a blackboard system

Algorithm 1 Norm Execution Process Based on Blackboard System

Input: *Initial State*: Initial information of Blackboard
Output: *Execute Path*: Executed norms

```
1  Control Shell Start;
2  Que ← TriggerEvent(NormList);
3  while NotEmpty(Que) do
4      Q ← Que;
5      while NotEmpty(Q) do
6          N = DeQueue(Q);
7          CreateSignal(N);
8          res = Action(N);
9          if PostCon(N) ∈ res then
10             ExecutePath.Add(N1);
11         else
12             ExecutePath.Add(N0);
13         ResCache.Add(res);
14     UpdateBB(ResCache);
15     Que ← TriggerEvent(NormList);
16 Control Shell Exit;
```

along with KSs and the blackboard. The control shell directs the problem-solving process by managing how KSs respond to contributions that are placed on the blackboard by an executing KS and to other events that may be triggered by the application or received from external sources. The *TriggerEvent* method (Line 2, Line 15) is proposed to detect norms which can be triggered by the information of blackboard. The *CreateSignal* method (Line 7) generates a signal to trigger the corresponding KS. The *Action* method (Line 8) returns the results of KS. We can get the expected results of a norm using *PostCon* method (Line 9). If the results of norm N are consistent with the expected results, it will add N_1 to the execution path. In contrast, N_0 represents its results are not coherent with the expectation of norm N. The *updateBB* method (Line 14) updates the information on the blackboard according to *ResCache*.

The MONE method is designed in three steps:

- To construct the LEG of the multiple norm execution tracks of the same agent;

Algorithm 2 Graph Construction

Input: N: a norm set, s_0: a starting state.
Output: $G \leftarrow (V, E, L)$: a LEG

1 $l_0 \leftarrow createLayer(s_0)$ //entrance;
2 $V \leftarrow \varnothing, E \leftarrow \varnothing, L \leftarrow \{l_0\}$;
3 **for** *each* $l \in L$ **do**
4 $s \leftarrow getState(l)$;
5 $S \leftarrow \varnothing$ as a temporary state set;
6 **for** *each* $r \in N$ *that* $r = a(t, e)$ **do**
7 **if** $\exists inState.t \sqsubseteq \exists inState.s$ **then**
8 $v \leftarrow createVertex(r)$;
9 **for** *each* $v' \in V$ **do**
10 $r' \leftarrow getNorm(v')$ as $a'(t', e')$;
11 **if** $\exists inState.e' \sqsubseteq \exists inState.t$ **then**
12 **if** $isMutated(v')$ **then**
13 $d \leftarrow createDashedE(v', v)$;
14 **else**
15 $d \leftarrow createSolidE(v', v)$;
16 $E \leftarrow E \cup \{d\}$;
17 $V \leftarrow V \cup \{v\}$;
18 $e'' \leftarrow getResult(a, s)$;
19 **if** $e \neq e''$ **then**
20 $isMutated(v) \leftarrow true$;
21 **else**
22 $isMutated(v) \leftarrow false$;
23 $t = getTime()$;
24 $s' \leftarrow updateState(s, e'', t)$;
25 $S \leftarrow S \cup \{s'\}$;
26 $mergeState(S)$;
27 **for** *each* $s \in S$ **do**
28 $L \leftarrow L \cup createLayer(s)$;

- To detect the pairwise mutations in the LEG;
- To evolve the norms related to the mutations.

We propose Algorithm 2 to construct the LEG for the different behaviors of the same agent on the norm tracks.

It starts with a special state s_0 that triggers at least a norm. A layer of conditions is constructed with the result of the action *starts*. The built-in methods are called to create a layer (Line 1), a vertex (Line 8), a dashed edge (Line 13) and a solid edge (Line 15). As a layer represents a state, and a vertex represents an executed norm, the *get* method returns a state (Line 4) and a norm (Line 10). The *getResult* method returns the effect of action a in the state s. If it contradicts to the expectation e of the norm r, the vertex v is marked as *mutated*. The *getTime* method (Line 23) returns the current time when the algorithm runs. It is added with the *updateState* function (Line 24) to the inherited conditions in s together with the effect e'' of the action, following Theorem 1.

Algorithm 2 creates the LEG for one time series as the layers are built sequentially with time elapses. In a scenario with multiple agents of the same type simultaneously, multiple such LEGs can be constructed. It is expected that all the norms are executed as expected, and all such LEGs are the same. But as is explained above, mutations may happen and there exist different edges even for the same type of agents with the same set of predefined norms. The expectations of the norms are compact and precise when defined to prescribe the result of the execution. The mutated result offers more information to evolve the norm definitions.

Algorithm 3 Mutation Detection

Input: $G \leftarrow (V, E, L)$: a LEG
Output: M: a set of candidate track αs

1 $M \leftarrow \varnothing, S \leftarrow \varnothing$;
2 **for** *each* $v \in V$ **do**
3 $Sign_m \leftarrow false, Sign_o \leftarrow false.$;
4 **for** *each edge* d *that* $v == getHead(d)$ **do**
5 **if** d *is dashed* **then**
6 $Sign_m \leftarrow true$;
7 **else**
8 $Sign_o \leftarrow true$;
9 **if** $Sign_m$ *and* $Sign_o$ **then**
10 $S \leftarrow S \cup \{v\}$;
11 **for** *each* $v \in S$ **do**
12 **for** *each* $v' \in (S - \{v\})$ **do**
13 $p \leftarrow createPath(v, v')$;
14 **if** $p \neq null$ **then**
15 $M \leftarrow M \cup \{p\}$;

Then Algorithm 3 is proposed to detect the mutations in the LEG.

As the vertices are merged representing the same norm, Algorithm 3 detects such vertices that have different results in execution processes. It queries for the vertices that have both a dashed edge (mutated) and a solid edge (obeyed) which represent the different effects of the same action. Then it tries to create a track between two such vertices (Line 13). Such tracks are used to evolve the norms as in Algorithm 4.

It takes a track that follows Theorem 2 as the input. The norm r on the tail of the track is the destination norm of the evolution. The algorithm takes the norm on the head (Line 2), verifies the norm execution through the track (Line 3) and *crossovers* the mutated norm with the destination one by one. The $\oplus$ operator (Line 7) takes all the conditions $\exists inState.e_i$, verifies if its negation can be deduced from t. The condition is added to t' if not contradictory to t.

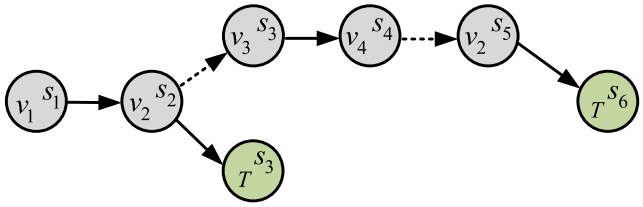
There could be other kinds of evolution strategies but this simple version is exploited to illustrate the applicability of Mone in this paper.

Although the reasoning complexity of the DL language is high, Mone does not rely on a common sense reasoning service. The complexity on Algorithm 3 is $\mathcal{O}(N_V N_E)$ where

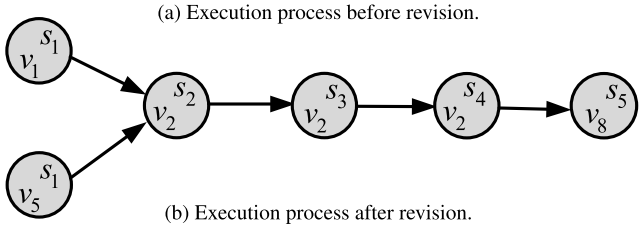
Algorithm 4 Norm Evolution

Input: p: a track
Output: r: the refined norm

```
r ← getNorm(getTail(p)) as a(t, e);
v_0 ← getHead(p);
for i ← 0; i < getLength(p) − 1; i++ do
    d ← getEdge(v_i, v_{i+1});
    if isDashed(d) then
        r_i ← getNorm(v_i) as a_i(t_i, e_i);
        t' ← t ⊕ e_i;
        r ← a(t', e);
```



(a) Execution process before revision.



(b) Execution process after revision.

FIGURE 4. Execution graph of the experiment.

N_V is the number of vertices and N_E is the number of edges in the LEG.

As far as we known from the state-of-the-art, there is no publications focusing on this kind of mutation detection and usage so we do not have a comparative experiment with other methods.

VI. VERIFICATION

We have designed a case study to validate the algorithms and simulate a scenario in sweeping robot system. Alice is a sweeping robot, and the interior of Alice is an agent-based system. Alice starts cleaning from $8:00AM$ every day if set in the Auto mode. But one day Alice was stuck by an obstacle, which resulted in the failure of the task. The agent-based system triggered a series of rules to detect problems. Figure 4a gives an example of the horizontal layered execution graph and the initial set of norms in our scenario are shown in Table 1.

There are two execution paths in Figure 4a. One series of execution is $(v_1, v_2, v_3, v_4, v_2, T)$, note that there is a dashed edge between v_2 and v_3, which means Alice didn't finish its work within limited time. So this series of execution indicates the robot is stuck in state s_2, the agent system changes to the monitor mode. After a series of detections, the sweeping robot executes r_2 normally and completes the cleaning task in state s_6. Another series of execution is (v_1, v_2, T) which

TABLE 1. The Information of norms in Figure 4.

Norm	Action	Trigger	expectation
N_1	$set\ Mode\ to\ Auto$	$Time \geq 8:00AM$	$Mode == Auto$
N_2	$get\ sweeping\ area$	$Mode = Auto$	$Mode = Working$
N_3	$set\ Mode\ to\ Check$	$Mode = Monitor$	$Mode = Check$
N_4	$obstacle\ detect$	$Mode = Check$	$Obstacle = True$
N_5	$remove\ obstacle$	$Obstacle = True$	$Obstacle = False$

TABLE 2. Information of execution nodes in Figure 4a.

Path	Level	Norm	Result	State
$Path_1$	s_1	r_1	$Satisfied$	$Mode = Auto$
	s_2	r_2	$Exception$	$Mode = Idle$
	s_3	r_3	$Satisfied$	$Mode = Check$
	s_4	r_4	$Exception$	$Obstacle = False$
	s_5	r_2	$Satisfied$	$Mode = Finish$
$Path_2$	s_3	r_3	$Satisfied$	$Mode = Check$

TABLE 3. Information of execution nodes in Figure 4b.

Level	Norm	Result	State
s_1	r_1	$Satisfied$	$Mode = Auto$
s_1	r_5	$Satisfied$	$Obstacle = False$
s_2	r_2	$Satisfied$	$Mode = Working$
s_3	r_2	$Satisfied$	$Mode = Working$
s_4	r_2	$Satisfied$	$Mode = Finish$

indicates the robot completes the work normally. There are four different types of nodes (including N_1, N_2, N_3 and N_1 in Table 1) shown in Figure 4a, the result of node v_2 in state s_2 and node v_4 in state s_4 is *Exception*; the result of other nodes is *Satisfaction*. The information of execution nodes is shown in Table 2. Preliminary test shows that mutations can be detected from the LEG as the two edges shown in Figure 4a.

- $e(v_2, v_3)$ mutates the norm that the task should complete but failed.
- $e(v_4, v_2)$ mutates the norm that the obstacle should be found and removed, but no obstacle is found.

In the execution track mentioned above, the norm r_2 in s_2 is finally executed correctly in another execution path, the sweeping robot successfully completes the task, therefore we can find some information that contribute to the evolution of the destination norm r_2. The norm r_4 mutated in state s_4, which means the system has not found the obstacle. As we mentioned in the earlier section, the mutated norm r_4 enriches the trigger of r_2 with no obstacle to be moved. In the part of norm evolution, the condition $obstacle = false$ is added to the trigger of norm r_2.

Mone enriches the trigger condition of the norm r_2 that mutates. The trigger of norm r_2 changed from $Mode = AutoClean$ to $Mode = AutoClean \wedge Obstacle = False$. One day, sweeping robot removed the obstacle which was found during cleaning work and therefore the robot will not stop working because of obstacles. The correct execution process of norms is shown in Figure 4b. The information for each norm is shown in Table 3.

The paper presents the example to revise norms based on the Mone framework. The crossover of the mutated norms with the norm on vertex 2 enriches the norm's trigger with

no obstacle to be moved. The experiments show that the mutations will be inherited by the well-executed norm and contribute to the evolution of the norm's trigger. The simulation experiment ensure our approach is able to detect all mutations and ensure that norms can be improved by some mutations.

VII. DISCUSSION

A. LOW-LEVEL NORM

In this paper, a description logic is used to represent the scenario of low level norms without considerations of temporal constraint, permission/obligation/prohibition, event, and many other features of modern BBS based applications. This is because the purpose of the paper is to illustrate the findings of mutations and its usage in the evolution of norms. Norms can be extended with more complicated formalisms to support practical context constraints in practical industrial applications. However, the mutations in complex norms still focus on the unexpected behaviors of the agent, which is contradictory to the expectation.

B. OUTER EVIDENCES

The layer in the LEG is used to incorporate the conditions for norm execution. In practice, there are outer evidences in the state of the layer that may lead to the mutations. But in Theorem 3 we assume no such outer evidences in a closed-world manner. This does not ruin the assumption of an open-world fundamental to the incompleteness of the norm predefinitions. The theorem works in the cases that the outer evidences do not contribute to the mutations.

C. CROSSOVER

The crossover is one of the key operators in the evolution theory. It is not studied thoroughly in this paper, only as a few lines of code in Algorithm 4. This is because the crossover function may be as simple as adding some conditions to the triggers of the norm, and may be as complex as to verify the relationships between the norms and locate the features of the norms to be fixed. The complex version of the crossover operator could be in the scope of the next paper.

VIII. CONCLUSION

In the flexible and open BBS, norms are defined on incomplete knowledge so the runtime executions of the norms may produce unexpected results. Researchers and engineers are always willing to fix such incompleteness with 'new' knowledge. However in the practical industry applications, various of features exist simultaneously and it is challenging to identify which is useful and how to use it in the norm evolution.

From our experiences of the BBS applications, the results of mutations are not always un-welcomed in the real world. With the detailed study of the different behaviors of the norms, it is found that mutations can be used as a source of the 'new' knowledge, to evolve the incomplete norm definitions. The Mone method has been proposed to record the execution tracks of the norms into a LEG, detect the mutations of the norms through the tracks, and crossover the mutated norm(s) with the norm to be fixed. Such evolution steps can enhance the completeness of the norms in a novel perspective.

REFERENCES

[1] D. D. Corkill, K. Q. Gallagher, and K. Murray, "GBB: A generic blackboard development system," in *Proc. AAAI Nat. Conf. Artif. Intell.*, 1986, pp. 1008–1014.
[2] M. J. North, E. Tatara, N. T. Collier, J. Ozik, and P. R. Corp, "Visual agent-based model development with repast simphony," in *Proc. Agent Conf. Complex Interact. Social Emergence*, Nov. 2007, pp. 173–192.
[3] N. Collier and M. North, "Parallel agent-based simulation with repast for high performance computing," *Simulation*, vol. 89, no. 10, pp. 1215–1235, Oct. 2013.
[4] X. Wang, H. Chen, Q. Zhao, and W. Li, "W—A logic system based on the shared common knowledge views," in *Proc. Int. Joint Conf. Artif. Intell.*, 2008, pp. 410–415.
[5] M. Fisher, R. H. Bordini, B. Hirsch, and P. Torroni, "Computational logics and agents: A road map of current technologies and future trends," *Comput. Intell.*, vol. 23, no. 1, pp. 61–91, Feb. 2007.
[6] N. Alechina, M. Dastani, and B. Logan, "Reasoning about normative update," in *Int. Joint Conf. Artif. Intell.*, 2013, pp. 1–5.
[7] E. Argente, G. Beydoun, R. Fuentes-Fernández, B. Henderson-Sellers, and G. Low, "Modelling with agents," in *Agent-Oriented Software Engineering*, M.-P. Gleizes and J. J. Gomez-Sanz, Eds. Berlin, Germany: Springer, 2011, pp. 157–168.
[8] M. El-Menshawy, J. Bentahar, W. E. Kholy, and R. Dssouli, "Verifying conformance of multi-agent commitment-based protocols," *Expert Syst. Appl.*, vol. 40, no. 1, pp. 122–138, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417412008858
[9] L. Gasparini, T. J. Norman, M. J. Kollingbaum, L. Chen, and J.-J. C. Meyer, "Còir: Verifying normative specifications of complex systems," in *Coordination, Organizations, Institutions, Norms Agent System*, V. Dignum, P. Noriega, M. Sensoy, and J. S. Sichman, Eds. Cham, Switzerland: Springer, 2016, pp. 134–153.
[10] F. Meneguzzi, O. Rodrigues, N. Oren, W. W. Vasconcelos, and M. Luck, "BDI reasoning with normative considerations," *Eng. Appl. Artif. Intell.*, vol. 43, pp. 127–146, Aug. 2015.
[11] W. W. Vasconcelos, M. J. Kollingbaum, and T. J. Norman, "Normative conflict resolution in multi-agent systems," *Auto. Agents Multi-Agent Syst.*, vol. 19, no. 2, pp. 124–152, Oct. 2009.
[12] E. A. Silvestre and V. T. da Silva, "Verifying conflicts between multiple norms in multi-agent systems," in *Proc. Int. Conf. Auto. Agents Multiagent Syst.*, Richland, SC, USA, 2015, pp. 2013–2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2772879.2773552
[13] J. S. Santos, J. O. Zahn, E. A. Silvestre, V. T. Silva, and W. W. Vasconcelos, "Detection and resolution of normative conflicts in multi-agent systems: A literature survey," *Auto. Agents Multi-Agent Syst.*, vol. 31, nos. 2–3, pp. 1–47, 2017.
[14] A. Kayal, W.-P. Brinkman, M. A. Neerincx, and M. B. V. Riemsdijk, "Automatic resolution of normative conflicts in supportive technology based on user values," *ACM Trans. Internet Technol.*, vol. 18, no. 4, pp. 1–21, Nov. 2018.
[15] L. S. Passos, R. Abreu, and R. J. F. Rossetti, "Spectrum-based fault localisation for multi-agent systems," in *Proc. 24th Int. Conf. Artif. Intell.*, 2015, pp. 1134–1140. [Online]. Available: http://dl.acm.org/citation.cfm?id=2832249.2832406
[16] C. M. Tang, W. K. Chan, Y. T. Yu, and Z. Zhang, "Accuracy graphs of spectrum-based fault localization formulas," *IEEE Trans. Rel.*, vol. 66, no. 2, pp. 403–424, Jun. 2017.
[17] Z. Huang and R. Alexander, "Semantic mutation testing for multi-agent systems," in *Eng. Multi-Agent Syst.*, M. Baldoni, L. Baresi, and M. Dastani, Eds. Cham, Switzerland: Springer, 2015, pp. 131–152.
[18] M. Mashayekhi, H. Du, G. F. List, and M. P. Singh, "Silk: A simulation study of regulating open normative multiagent systems," in *Proc. Int. Joint Conf. Artif. Intell.*, 2016, pp. 373–379.

XIANCHANG WANG received the Ph.D. degree in computer science and engineering from the National University of Defense Technology, China, in 1991. He is currently a Professor with the College of Computer Science and Technology, Jilin University, China. He has published innovative articles in journals and conferences, such as ICLP, IJCAI, *Journal of Artificial Intelligence*, and the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING (TKDE). His research interests include artificial intelligence and multi-agent systems.

RONGHAO FU received the B.S. degree from the College of Computer Science and Technology, Northeast Petroleum University, in 2018, where he is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Jilin University, China. His current research interests include artificial intelligence and multi-agent systems.

RUI ZHANG received the Ph.D. degree. He is currently an Associate Professor with Jilin University, China. He has published a book *Relation-Based Access Control* (Springer aka Verlag, Germany), and 15 articles including two CCF B and five SCI indexed journals. His research interests include knowledge representation and data integration.