Fall 2019

The Generation of Operational Policy for Cyber-Physical Systems in Smart Homes

Jared Wayne Hall
Missouri State University, Jared007@live.missouristate.edu

THE GENERATION OF OPERATIONAL POLICY FOR CYBER-PHYSICAL SYSTEMS IN SMART HOMES

A Master's Thesis

Presented to

The Graduate College of

Missouri State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science, Computer Science

By

Jared Wayne Hall

December 2019

THE GENERATION OF OPERATIONAL POLICY FOR CYBER-PHYSICAL SYSTEMS IN SMART HOMES

Computer Science

Missouri State University, December 2019

Master of Science

Jared Wayne Hall

ABSTRACT

The term “Cyber-Physical Systems” (CPS) refers to those systems which seamlessly integrate sensing, computation, control, and networking into physical objects and infrastructure [1]. In these systems, computers and networks of physical entities interact with each other to bring new capabilities to traditional physical systems. Since its introduction, the field of Cyber-Physical Systems (CPS) has evolved with new and interesting advancements concerning its capability, adaptability, scalability, and usability [1]. One such advancement is the unification of the Internet of Things (IoT), a concept that enables real-world everyday objects to connect to the internet and interact with each other, with CPS [1]. This emerging technology (CPS/IoT), however, comes with its own set of challenges. One such challenge is the need for new technologies to enhance the usability of CPS/IoT systems in smart homes by making it easier for users to manage and develop the core operating logic these systems employ. This need arose as smart home CPS/IoT systems have become more ubiquitous, with broader environments and more complex architectures and yet, they still rely on handwritten rules or hard coding the operational logic into the system. In light of this challenge, we introduce the concept of “operational policy”, a plan for the operation of a CPS/IoT system and frame the task of building the operational policy as a combinatorial optimization problem that algorithms can solve. We then introduce “Associative-Behavioral” rules, which constrain human input in the rule building process to selecting the actions they want their devices to take in a given scenario. From here, we introduce two algorithms for the generation of operational policy: an adaptation of the Differential Evolution algorithm and the A-Posteriori algorithm. These algorithms are compared with a brute force method of policy generation in a simulated CPS containing 60 devices with a focus on usability. We found that in a situation where there is previous information on the CPS, A-Posteriori performs the best. Whereas for the mid to large scale scenarios in our simulation without previous information, Differential Evolution performs the best. And finally, in small scale scenarios without previous information brute force was best.

KEYWORDS: cyber-physical systems, internet of things, operational policy, policy generation algorithms, associative behavioral rule generation, smart homes

THE GENERATION OF OPERATIONAL POLICY FOR CYBER-PHYSICAL SYSTEMS IN SMART HOMES

By

Jared Wayne Hall

A Master's Thesis
Submitted to the Graduate College
Of Missouri State University
In Partial Fulfillment of the Requirements
For the Degree of Master of Science, Computer Science

December 2019

Approved:

Razib Iqbal, Ph.D., Thesis Committee Chair

Anthony Clark, Ph.D., Committee Member

Jamil Saquer, Ph.D., Committee Member

Lloyd Smith, Ph.D., Committee Member

Julie Masterson, Ph.D., Dean of the Graduate College

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

INTRODUCTION

The term “Cyber-Physical Systems” (CPS) refers to those systems which seamlessly integrate sensing, computation, control, and networking into physical objects and infrastructure [1]. Formally, a CPS consists of three components: A cyber component that performs computation (e.g., information processing and control), a communication component that handles the communication between the cyber and physical components, and a physical environment that consists of physical entities or processes [1, 2] as shown in figure 1 below. In these systems, the cyber component is tightly coupled with the physical component via a feedback loop [1, 3] where the physical environment informs the cyber component of its current state via the communication component, and the cyber component exerts control over the physical environment which alters its state, causing the cycle to repeat.

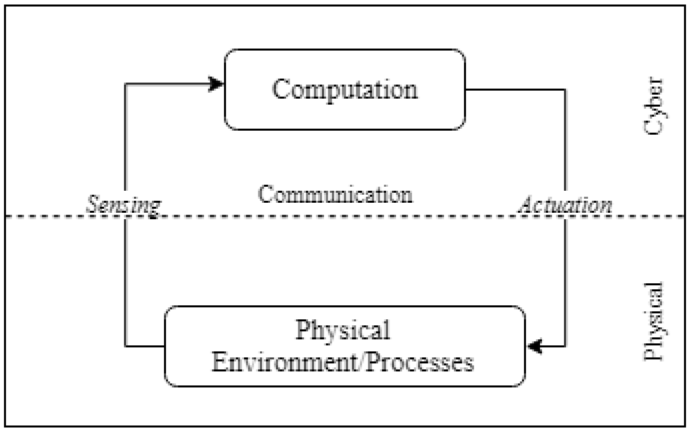

Figure 1: Cyber-Physical Systems Component model

Since its introduction in 2006, the field of Cyber-Physical Systems (CPS) has evolved with new and interesting advancements concerning its capability, adaptability, scalability, and usability [1]. One such advancement is the unified field paradigm of the Internet of Things (IoT) with CPS, which introduces CPS/IoT systems [1]. The IoT is a concept that enables real-world everyday objects to connect to the internet and interact with each other. In the CPS/IoT paradigm, the two fields are essentially merged with the following changes:

- The IoT takes the place of the communication component in the traditional CPS model.
- The CPS/IoT software platform/middleware is called by the generic term "platform".
- The platform and the physical environment are wrapped together and given the term "CPS/IoT system".

In this paradigm, humans play a central role both as entities in the physical environment and as users of the system (e.g., "human-in-the-loop") via their interactions with both the system and the physical environment. Because of this, humans are added as the fourth component in CPS/IoT systems, as stated by the National Institute for Standards and Technology (NIST) in [1]:

> "… reflect the varying roles humans may have in CPS/IoT systems, ranging from user to component, environmental factor, etc. (for example, for a Level 3 automated vehicle a passenger is a user, a safety driver is a component, and a pedestrian is an environmental factor). The interactions of humans with CPS/IoT systems may be limited to the logical realm, to the physical realm, or extend to (and link) both. Because of this diversity of interactional modes, humans are treated as a distinct component in the CPS/IoT Components Model."

This emerging technological paradigm, however, comes with its own set of challenges. One such challenge is the need for new technologies to enhance the usability of CPS/IoT systems in smart homes by making it easier for users to manage and develop the core operating logic these systems employ. As CPS/IoT systems continue to advance, the complexity of the operational logic needed to manage their physical environments has grown to the point where hardcoding operational logic into the system has become a significant burden. This need arose as both CPS and IoT platforms, and consequently, CPS/IoT systems, have become more ubiquitous, with broader environments and more complex architectures and yet, they still rely on handwritten rules or hard coding the operational logic into the system as will be discussed in the literature review. In general, hardcoding the logic or providing handwritten rules limits both the practical scale of any potential CPS/IoT platform and the general base of users to only those users with high levels of knowledge in rule design and the domain of application for the particular CPS/IoT system.

One potential solution for this challenge would be "human-in-the-loop" algorithms that generate the complex logic that CPS/IoT systems need in order to operate their physical environments. These algorithms would enable the system to decouple the decision-making process from the process of defining the core operational logic and shift this process to an auxiliary system designed specifically for CPS/IoT operational logic generation; increasing the efficiency of the system itself. The use of such algorithms would also increase the usability and security of the system since these algorithms would allow users to specify the behaviors they want without requiring them to write the rules themselves or giving them access to the underlying code. For example, consider the typical process to define the behavior of a CPS/IoT system that operates a smart home. First, the user would need to either access the systems underlying code and manually program the behavior they want themselves or they would need to handwrite the rules and inject them into the system to see its behavior change. This presents a barrier to the adoption of CPS/IoT systems since it requires a lot of knowledge from the user (in programming and in the systems software architecture). In contrast, an algorithm that guides the user through the process of defining the operational logic for the CPS/IoT system and simplifies it to where the user just selects the action(s) they want for a given environmental scenario would make the system more usable. To the best of our knowledge and after conducting background research into a variety of CPS/IoT proposals, we have not been able to find any human-in-the-loop algorithms that generate operational logic for CPS/IoT systems.

Considering points mentioned above, for this thesis, we investigated the problem of operational policy generation for CPS/IoT systems and proposed both an adaption of the differential evolution algorithm from the field of evolutionary computing and a novel algorithm, the A-Posteriori algorithm, to generate an operational policy for CPS/IoT systems.

LITERATURE REVIEW

Over the past decade, as research on the IoT and CPS have converged to form the CPS/IoT paradigm, the operating logic for CPS/IoT systems has evolved [1]. During our background research into the subject of rule generation for CPS/IoT platforms, we have noticed a sectional delineation of the methods that proposed CPS/IoT architectures use to manage the core operating logic of their systems. This delineation takes the form of two major groups: systems that hardcode the logic and systems that use rules to soft-code the logic.

Hardcoded Operational Logic in CPS/IoT Systems

Early systems focused rather exclusively on the use of hardware-based architectures that heavily relied on simple sensors, such as RFID tags, and simple actuators, such as a light or a gate, for the implementation of the core CPS/IoT vision such as the systems presented in [4-7]. The system in [4] is a classic example of many these kinds of CPS/IoT systems. These systems typically feature some hardware-based solution for interacting with a limited set of device types, such as sensors, actuators, Radio Frequency Identification (RFID) tags, or Machine to Machine (M2M) devices. Accordingly, the operational logic for these types of systems primarily consists of a handwritten programming script in the authors chosen programming language. Since this logic is manually programmed into the system, changing the way these CPS/IoT systems operate is expensive as the only way to change the operational logic of the system in this paradigm is to turn it off and alter its internal code. Additionally, all CPS/IoT systems utilize control engineering [8] in the design of the CPS/IoT system itself as well as its physical entities. However, hardcoded CPS/IoT systems leave it to the user to manually alter the engineered system in order to introduce new plants (entities) or to alter the systems processes or variables

[8]. This becomes a problem due to the lack of reusability and elasticity in these CPS/IoT systems as any changes or growth in these systems would require the user to turn them off and re-engineer the system.

To begin addressing this problem in the field, researchers began developing and introducing a plethora of technologies aimed at reducing the level of expertise required to provision and set up a CPS/IoT system and improve their level of reusability and elasticity. For instance, in [9], the authors present the concept of "object virtualization", where a physical object is represented in the virtual world with its respective properties. Object virtualization enables the CPS/IoT system to represent entities in its physical environment with a standard interface or abstraction. Therefore, instead of needing to write code to manipulate each physical entity in that entity's language, an arduous task, the user can instead use a standardized method for defining the entities behavior [9]. Another example of these technologies that increase the usability of CPS/IoT systems would be [10], which introduces a cloud-based CPS/IoT system with discrete pre-programmed objects that the user simply needs to select instead of needing to program the system themselves.

A third example would be the advent of CPS/IoT recommender systems [11]. According to [12], A recommender system (i.e., recommendation system) "is a subclass of information filtering system that seeks to predict the "rating" or "preference" a user would give to an item or service". These systems are already ubiquitous from movie and social media recommendations to systems that recommend software products [11]. In CPS/IoT these systems often focus on recommending items [13] or services [14] to the user and can be used to ease the information overload in trying to get the right devices or services for that user's CPS. Some recommender systems, such as the one presented in [15], can even recommend devices or services from across multiple application domains.

Soft-Coded Operational Logic in CPS/IoT Systems

Among the technologies introduced to make using CPS/IoT systems easier to use, one of the most influential [16, 18] would be the introduction of the event-driven rules model to CPS/IoT systems such as the one presented in [17]. These rules add artificial intelligence into the operating logic of the CPS/IoT system and simplify the cost of building the physical environment (i.e., the time and effort) by allowing the user to simply write a set of rules to govern the CPS/IoT system and then leaving the enforcement of said rules to the system [16]. These rules come in the form of Event-Condition-Action (ECA) rules. According to [19], an ECA rule: "autonomously reacts to actively or passively detected simple or complex events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happens, and the condition is true". Formally, ECA rules consist of independent statements in the following format: **ON** Event **IF** Condition **DO** Action. In CPS/IoT, an event is generated by "entities" or "things" in the physical environment (e.g., the temperature is 75 degrees). A condition is a Boolean trigger constraint on the rule itself (i.e., the rule will only fire if the condition is true). Whereas, an action is a control to be enforced on the entity the rule applies to (e.g., turn on the fan). Some notable characteristics of ECA rules, as stated in [17] are:

- A rule is activated only by events
- Its execution is autonomous and independent of other rules in the system
- It implements a reaction to the incoming event
- It contains a guarding condition to execute such actions

A crucial benefit of using rules instead of programming scripts to control the core operating logic of a CPS/IoT system is that they can be easily grasped by individuals with no background in computer science as ECA rules are already in common use by many application domains [18]. This ubiquity expands the user base of CPS/IoT systems while also giving the engineers of these platforms the ability to decouple the core operating logic from the enforcement system since rules can be altered on the fly without giving the user access to the underlying code.

The CPS/IoT systems presented in [20-22] are representative examples of many CPS/IoT systems that utilize ECA rules. This kind of approach is widely used in the IoT industry with platforms such as Amazons IoT platform (https://aws.amazon.com/iot/) and Xively (https://www.xively.com/) and in CPS/IoT systems research such as is seen in [16-18, 20-22]. These systems come with a number of benefits, such as, device virtualization, as explained in [9], an elastic cloud architecture that allows users to change/create building blocks in their service definition on-demand [23]. They also contain the ability to dynamically alter the ruleset for each system by allowing the user to manually write the rules and inject them into the CPS/IoT system. These kinds of systems were the first to feature a decoupling of the operational logic from the code itself, allowing users to change the system on the fly increasing the elasticity of the CPS/IoT system itself. These types of systems often come with dynamic rule engines or other artificial intelligence agents [18], that decouple the operational logic from the system itself giving it robust elasticity and while also making the platform itself reusable for different application domains. Though the idea of generating operational policy has not been addressed, these types of CPS/IoT systems will reap the most benefit from this research.

Throughout our background research into IoT/CPS systems, we have not been able to find systems that offer the ability to generate operational policies for their environments using a human-in-the-loop algorithm. Currently, there are many systems such as those seen in broad surveys of the field, [24-25, 27], that do not use rules but instead hardcode the operational logic via programming scripts. Of the systems that do use rules, the generation of the rulesets is done entirely manually, such as in [20-22]. To the best of our knowledge, there are no current CPS/IoT systems that offer the ability to generate their rule sets via a human-in-the-loop algorithm. As mentioned in the introduction, this poses a problem as the scale of CPS/IoT systems grows, and it is our belief that the application of the idea of "Operational Policy" to CPS/IoT has the potential to solve these problems.

OPERATIONAL POLICY FOR CPS/IOT SYSTEMS

In the most general sense, a “policy” is a step-by-step plan for the operation of a system. The policy maps out how a system should behave in a given context, at a particular moment in time, and what constraints it should obey [26]. Thus, the application of the idea of policy to the domain of CPS/IoT systems in smart homes is particularly useful. If one could generate a policy for a given CPS/IoT system beforehand, then that systems core job would be greatly simplified. This is because the CPS/IoT system can now focus solely on enforcing the current policy and leave the tasks of rule classification and generation to another process. This should result in increased efficiency in the CPS/IoT platform’s runtime processes.

Considerations on Policy for CPS/IoT Systems

However, before we apply the concept of policy to CPS/IoT systems, we need to consider the following:

Standardized Policy Format. Another benefit of using policy for the management of a CPS/IoT systems’ physical environment is the interoperability of multiple policies for a given configuration. For instance, a CPS/IoT system that is managing a power grid may have one policy for standard operating procedure and another policy for emergency situations. This interoperability between different policies allows the given CPS/IoT system to be adaptable without needing to re-provision the CPS/IoT platform or rewrite its code in order to change its behavior. However, in order to do this efficiently, we would need a standard format for the policy itself so that potential users of the CPS/IoT platform can use these policies without needing access to the platform's code.

Entity Interaction and Dependencies. A policy must take into consideration that the entities within the CPS/IoT systems' physical environment may regularly interact with each other. Sometimes a particular entity may depend on communication from another entity as a condition to perform an action, while other entities, such as sensors, may simply exist to provide environmental information such the temperature or the humidity. There is a potential for dense interconnectivity in CPS/IoT systems where the actions a given entity should execute depends on the current state of another. For instance, in our smart home setting, an alarm should only sound if the homes security sensors detect an intrusion. Since, as mentioned earlier, a policy is a step-by-step plan for the operation of a system, it must be able to support this level of communication.

Consideration of Physical Environment. Since the policy will be used to manage entities in a real-world environment via a CPS/IoT system, a valid policy must take this environment into consideration. It cannot require devices to perform actions that would be detrimental to the health of that environment or are impossible to perform (e.g., telling a door that is closed to close again, telling a limited hinge to move past its limit, etc.).

User Satisfaction. For all CPS/IoT systems, during their initial setup, a person must define and approve of its core operational logic. For the purpose of operational policy generation, we call this person the "user" in the policy generation dynamic. Since this user is the ultimate arbiter in whether or not any policy is valid, user satisfaction must be taken into consideration when generating said policy. This requires the use of user input to be able to determine user satisfaction with a given policy.

Many CPS/IoT systems are built to manage physical environments that humans inhabit, whether that be in public, the workplace, or at home [27, 28]. Because humans are, by nature, social creatures, any realistic attempt to automate the environments that humans exist in must take this social nature into account. After all, different people will want their respective environments to operate in a myriad of different ways. Additionally, given the same user and different social or environmental contexts, they may want the same environment to operate in very different ways. This principle extends across all CPS/IoT application domains. Thus, any realistic application of policy to CPS/IoT systems must consider the consistent and potentially vast changes to the operation of the system caused by natural human expectations driven by specific social or environmental contexts.

For instance, consider two friends: Sally and Martha. The subject is the temperature control of Sally’s smart home. When Sally is home, perhaps she likes the temperature to be 60 degrees Fahrenheit with 35% humidity, but Martha likes the temperature and humidity to be around 70 degrees Fahrenheit and 50% humidity. Therefore, when Sally is home by herself, she expects the environment to adapt to her wishes. However, when Martha visits, maybe she wishes the environment to change to be more socially acceptable by increasing the temperature and the humidity. This example is also subject to environmental context as well. For instance, to keep this temperature schema consistent during the summer, the home must manage the air conditioner since the outside environment is the main source of heat to the home. Whereas during the winter, the home would manage the heater or furnace instead since the outside environment has an effect of cooling the home. A second example would be the door lock into Sally’s smart room: when she is gone, perhaps she wants the door to remain locked no matter what. When she is home, and it is nighttime, she wants the door to only ring the doorbell when a person approaches the door. During the day, she wants the door to unlock when a person on her friend's list approaches otherwise just ring the doorbell.

These scenarios make apparent the social contexts with which we make decisions about how our environments should operate. This factor, that people change their minds on what's acceptable over time and context, expands to the smart city as well with the city's infrastructure needing to be plastic to the differences in the time of day, the weather, the time of year, a sudden crisis, or social events just to name a few factors.

The Definition of "Operational Policy"

To answer the considerations discussed in the previous subsection, we are introducing the idea of an 'Operational Policy". An Operational Policy is defined as a collection of rules which implement the higher ideal of policy for every entity that can perform actions within the CPS/IoT systems' physical environment on a step by step basis. A behavioral rule is a simple $IF\ conditions\ THEN\ action$ (e.g. $conditions \rightarrow action$) statement that maps some environmental conditions a given actor in the environment may encounter to an action. A collection of these rules would then form a schema (i.e., a plan) for how an acting entity should behave when given various environmental conditions (i.e., contexts). Given the considerations regarding the application of policy to CPS/IoT systems discussed in the previous subsection, we define a valid operational policy as a collection of behavioral rules that solve the following problem statement:

Given a CPS/IoT system whose physical environment consists of the following: 1) A non-empty superset of entities, where each member contains a set of discrete states and may also contain sets for its actions, and other entities it interacts with. 2) A set of discrete, measurable states for the physical environment as a whole were each state is a snapshot of the states of each entity in the set of entities. And 3) a measurement of user satisfaction. Find an operational policy, a collection of rules for each entity that can perform actions based upon either that entity's state alone or said entity's states along with the states of its interactive neighbors, that maximizes the user's satisfaction in the operation of the CPS/IoT systems' physical environment without violating the entity's operational constraints.

To state this problem mathematically; Given the following universe:

- A set of entities $\boldsymbol{E} = \{e_1,\ e_2, \dots ,\ e_m\}$, where each entity, at a minimum, contains a set of discrete states $\boldsymbol{S} = \{s_1,\ s_2, \dots ,\ s_m\}$ and the variable $S_{current}$, where $S_{current}\ \in \boldsymbol{S}$. Each entity may also contain the following additional sets:
    - a set of actions $\boldsymbol{A} = \{a_1,\ a_2, \dots ,\ a_m\}$
    - a set of other entities in the environment it interacts with $\boldsymbol{I}\ = \{i_1,\ i_2, \dots ,\ i_m\}$, where $i_i \in \boldsymbol{E}$.
- A physical environment that can be quantified as the set of all possible entity states $\boldsymbol{Env} = \{Env_1,\ Env_2, \dots ,\ Env_m\}$, where each $Env_i\ =\ \{x\ |x = \bigcup_{i=0}^{n} \boldsymbol{E}_i[S_{current}]\}$, and a variable $Env_{current}$, where $Env_{current}\ \in \boldsymbol{Env}$.
- A user U whose satisfaction is $-1\ \leq\ \boldsymbol{U}[\boldsymbol{Env_{current}}] \leq 1$

Find an Operational Policy OP such that the following formulas are true:

$$\forall \boldsymbol{E}, where\ \boldsymbol{A} \neq \emptyset, \exists \boldsymbol{R}\ such\ that\ \boldsymbol{R} = \{X \rightarrow Y | X \in \boldsymbol{S} \vee \mathrm{X} \in \boldsymbol{I_S} \wedge Y \in \boldsymbol{A}\} \qquad (1)$$

$$\boldsymbol{OP} =\ \{\forall \boldsymbol{Env}, \exists \boldsymbol{R}\ \mid \boldsymbol{R}(\boldsymbol{Env_{current}}) \xrightarrow{yields} actions, where\ \boldsymbol{E}(actions) \xrightarrow{yields} Env_{next}\ , \qquad (2)$$
$$and\ \boldsymbol{U}(\boldsymbol{Env}_{next})\ is\ maximised$$

This problem falls under the class of problems known as combinatorial optimization problems. These problems consist of finding an optimal solution that consists of a subset of a finite set of objects, as discussed in [29]. For large problems, an exhaustive search is not tractable. Such problems operate in the domain of those optimization problems, in which the set of feasible solutions is discrete or can be reduced to discrete, and in which the goal is to find the best solution. Of course, optimization problems have a long history in CPS/IoT, just as they do in other Computer Science and Engineering disciplines. Many of the papers that we have seen focus on the optimization of the core CPS/IoT architecture, such as in [30, 31], or on the optimization of control enforcement as is done in [32]. However, we have not found any papers that focus on the optimization of the operational logic of CPS/IoT systems to human specifications.

ON ASSOCIATIVE-BEHAVIOURAL RULES AND THEIR GENERATION

As discussed in the previous section, the overall goal of operational policy generation is to create a set of context-sensitive rules for governing the operation of CPS/IoT systems in smart homes. However, as discussed in [26, 33], the application of rules leads to a number of issues such as policy conflicts and increased computational overhead for rule selection and evaluation. These issues primarily occur due to the structure of the rules and how they are used in CPS/IoT. Policy conflicts occur when we have multiple rules for the same event with overlapping or ambiguous conditions. For instance, if we have one rule for a fan that tells it to be on when the user is at their desk and another rule that says to turn off if the temperature is below 65 degrees, then we would have a conflict if the user is at their desk and the temperature is below 65 degrees. The increased computational overhead occurs when the rules are evaluated as each rule's conditions are evaluated linearly. This means that the rule engine needs to evaluate every condition in each rule before triggering the rule, which leads to extra computations being done. This is not ideal in a CPS/IoT system since rules may have many conditions.

In an effort to address these issues, we introduce the Associative-Behavioral (AB) rule. The AB rule offers further expansions to the behavioral rule model by merging the concept of event-specific triggers from event-condition-action (ECA) rules and the structure of behavioral rules with an efficient schema for rules with many conditions. The AB rule was developed to serve as a special application of the commonly used rules-based model for CPS/IoT physical environments that consist of networks of entities with discrete, manageable states. AB rules are designed for the management of the various entities in the CPS/IoT system's physical environment and are written from said entity's perspective. Below, we present a formal definition of AB rules when given the following:

- A set of entities $\boldsymbol{E} = \{e,\ e_2, \dots,\ e_m\}$, where each entity, at a minimum, contains a set of discrete states $\boldsymbol{S} = \{s_1,\ s_2, \dots,\ s_m\}$ and the variable $S_{current}$, where $S_{current} \in \boldsymbol{S}$. Each entity may also contain the following additional sets (The sets A, and I may be empty):
    - a set of actions $\boldsymbol{A} = \{a_1,\ a_2, \dots,\ a_n\}$
    - a set of other entities in the environment it depends on, (i.e., "Linked Entities"), with $\boldsymbol{I} = \{I_1,\ I_2, \dots,\ I_n\}$, where $I_n \in \boldsymbol{E}$.

The Associative-Behavioral Rule **AB** is defined as (3) for any entity that can execute actions:

$$\boldsymbol{AB} = \left\{X \rightarrow Y \middle| X = \left\{\boldsymbol{S}_{current} \wedge \cup_{k=0}^{|\boldsymbol{I}|} \boldsymbol{I}_k(\boldsymbol{S_{current}})\right\} \, and \, Y \in \boldsymbol{A}\right\} \tag{3}$$

There are some significant differences between typical behavioral rules, ECA rules, and AB rules to be noted here: First, instead of dealing with general events built by the entities themselves, for example an event generated when a door sensor transmits a message, AB-Rules deal with a specific kind of event that strictly represents the current state of a given entity (e.g., the state of the door sensor itself – high, low, off). This specification exists because of the domain this type of rule is designed for. In the operation of CPS/IoT systems, the most important information an entity in the system's physical environment can give is their states and available actions. Therefore, this information about the states and actions is what the rule is built from.

In accordance with this, the condition to trigger (e.g., its trigger condition) the rule is specifically that the entities the rule applies to have all reached a particular state. This principal includes any networked entities. These networked entities are called "linked entities" because the action that the current device is supposed to take is dependent on these linked entities reaching a certain state. For instance, consider the smart door in the home illustrated in the section on operational policy. The smart door is self-locking and comes equipped with a motor system to open and close itself, a microphone to allow users to open the door by issuing verbal commands such as “open the door” or “close the door”, and a camera to identify users. In this scenario, the information required for the door to know what action it should take is dependent on the current states of its linked entities, namely the camera and the microphone.

Second, these rules are only written with respect to the entities that possess the ability to execute actions. This distinction necessitates a sub-category for all entities in any CPS. Therefore, those entities that can execute actions are labeled “actuators”, while the entities that cannot execute actions and only provide information regarding their states are called “sensors”. This categorization is a bit broader than what is typically considered a sensor or actuator in engineering. In this sense, any entity in the known universe that can be told to execute a discrete action by a CPS/IoT platform is classified as an actuator. For instance, a phone, a door, a light, even robots or other software systems present in the CPS/IoT platform’s field of control. It should be noted that the only difference between a sensor and an actuator in our definition is that actuators possess the ability to execute actions when told to do so by the CPS. Of course, both sensors and actuators possess measurable states. But a sensors sole job is to provide that information, while an actuator can also execute actions.

Network Motifs in CPS/IoT Systems

The distinction between sensors and actuators, mentioned above, in operational policy generation when combined with the principle that only the actuators are considered during policy generation allows us to use what is known as a “network motif” when discussing any given actuator and its linked entities whether they be sensors or other actuators. For the purpose of our research, we are using the standard definition of a network motif used commonly throughout network theory as defined stated in [34]:

"All networks, including biological networks, social networks, technological networks (e.g., computer networks and electrical circuits) and more, can be represented as graphs, which include a wide variety of subgraphs. One important local property of networks are so-called network motifs, which are defined as recurrent and statistically significant sub-graphs or patterns. Network motifs are sub-graphs that repeat themselves in a specific network or even among various networks. Each of these sub-graphs, defined by a particular pattern of interactions between vertices, may reflect a framework in which particular functions are achieved."

In our application domain, a network motif consists of a given actuator and any linked entities, as shown in figure 2 below. In every one of the motifs shown in figure 2, the general pattern is the same: a single actuator (i.e., a sink in our sub-network) possesses zero or more linked entities. These linked entities can be either sensors or other actuators.

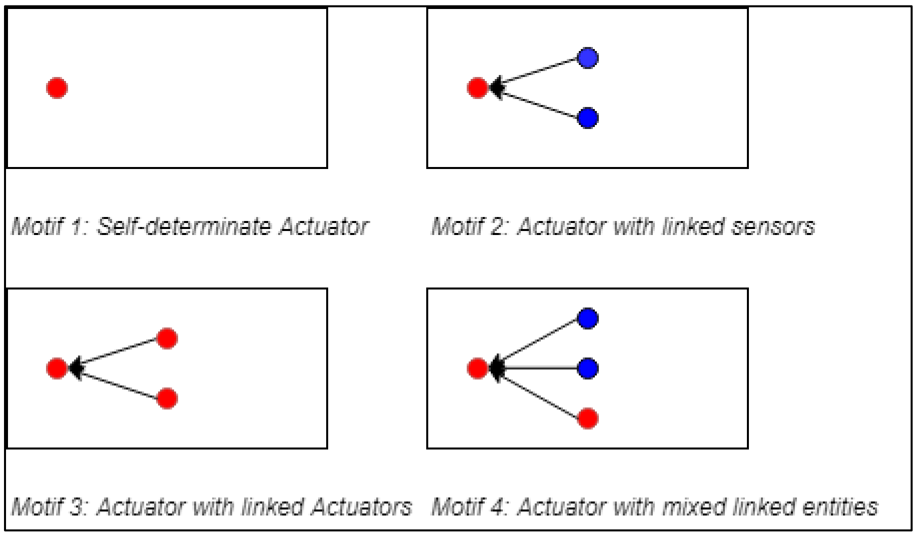
Motif 1: Self-determinate Actuator
Motif 2: Actuator with linked sensors
Motif 3: Actuator with linked Actuators
Motif 4: Actuator with mixed linked entities

Figure 2: Network Motif examples

To elaborate, consider the following examples: Motif 1 contains a self-determinate actuator. An example of this would be a pace-maker that relies upon its own information to know what to do next. Motif 2 contains an actuator with linked sensors. An example of this would be an air conditioning system with linked temperature sensors. In motif 3, we see an actuator with linked actuators. An example of this would be a cooling fan adjacent to a stovetop and an oven with a policy that If the stovetop or the oven is on, then the fan turns on. In motif 4, we see an actuator with mixed linked entities. An example of this would be a factory assembly line with robotic arms and a conveyor belt. This factory could have a policy such that if the belt is on and an item to be worked on is in front of the robot, then the arms script to perform its core job is activated.

The use of network motifs allows us to generate the rules needed for every entity in the CPS/IoT systems field of control, (i.e., its "environment") by generating rules for each motif in the network. However, unlike in typical applications of network theory, such as [22], finding these motifs is trivial. To do so, a prospective policy generation algorithm just needs to iterate through all entities in the network that possess actions (i.e., the actuators). One thing to note is that sensors cannot be the sink node in these subgraphs. This is because sensors cannot execute any actions. Therefore, it would make no logical sense to generate rules for them and by extension, create motifs around them.

For AB rules, we have given our network motifs a standard notation form, as shown below in formula (4). We can think of formula (4) as saying: "The states of the linked entities 1 through n influence/determine what actions the actuator should take". This influence is what constitutes the primary communication in our CPS/IoT system.

$$Linked\ Entity\ 1, \ldots, Linked\ Entity\ n\ \rightarrow Actuator \tag{4}$$

For example, consider the following scenario: Given our smart door from the previous section but with an added WIFI light linked to the door and the rule "if the door opens then turn on the light". This scenario would spawn a network, as seen in figure 3 below. In this scenario, there are two network motifs: $Cam, Mic\ \rightarrow Door$ and $Door\ \rightarrow Light$. This is to say that 1)

Based on the camera, microphone, and the doors states the door will execute some actions, and 2) based off the state of the door and the light, the light will execute some actions. To find out what actions both will execute, we just need to reference the rules: “if the door opens then turn on the light” and “if the user is at the door and says open, the door opens”.

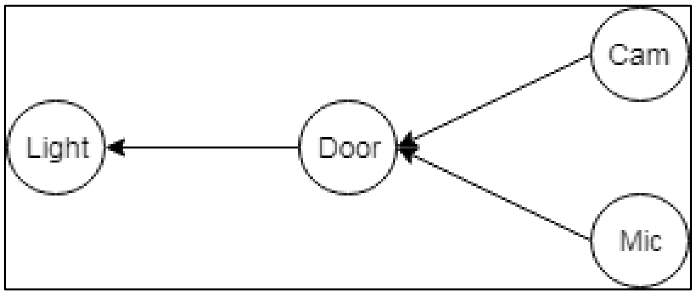

Figure 3: Network Motif for the Smart door Scenario

In this specific case, the network motifs do a few major things for us: First, they give us the list of entities that we must use to create the operational policy for this CPS/IoT system’s physical environment (i.e., the door and the light). Second, they provide a consistent method for the organization of our rules. Finally, they enable us to take advantage of the concept of the “Cohesive State”, which effectively solves the issues of computational overhead involved in rule selection and evaluation, as well as providing a safe manner to handle conflicting rules.

The Cohesive State

As discussed earlier, for any particular rule to be triggered its associated actuator must reach a certain state and that actuators linked entities, represented in its network motif, must also reach the states described in the rule’s trigger condition. This provides a discrete set of conditions for the rule to be triggered. Because of this principle, there are some qualities of the

AB rule format and network motifs that can be taken advantage of to greatly increase the speed of rule selection and evaluation.

First, each of the conditions in the AB-rule can be simplified to produce a discrete combination of states called the "Cohesive State". To get a cohesive state from a given network motif, we replace the entities in the network motif with their current states. For our smart door scenario, this would provide us with the notation shown below in (5):

$$< cam\ state >, < mic\ state > \rightarrow < door\ state > \tag{5}$$

However, by itself, this does not give us all of the information that we need to build a rule. If we recall the format for rules, in general, is **IF** conditions **THEN** action, then there is a little work that must be done to get our motif format into rule format. First, we must move the actuators state to the side with the conditions; then we will put the action to be executed in the place where the actuator state was. This is because, in addition to the linked entities current states, we also need to know the current state of the door itself as it would be redundant to tell an open door to open. These actions yield the following AB rule shown in (6) below:

$$< door\ state >, < cam\ state >, < mic\ state > \rightarrow < door\ action > \tag{6}$$

$$closed - locked, user - present, open\ \rightarrow open\ the\ door \tag{7}$$

In this case, there is an implicit "and" relationship between all of the states. This is to say that the door must be a certain state, such as closed, and both the cam and the microphone must also be at a certain state as shown in the sample rule given in (7) . This set of conditions at the left side of the rule is what we refer to as the “Cohesive State” and is an exceptionally powerful concept in regard to both policy generation and to rule selection/evaluation. What the cohesive state enables us to do is to reduce the process of rule selection to the simple character hashing done commonly in associative arrays instead of evaluating each condition to trigger the rule as is done currently. This allows the cohesive state to be used as a key to the specific action that the device must take when that cohesive state is encountered. To do so, we begin by generating each applicable cohesive state and save them in a hash table or associative array with an associated action. Then, we can trigger any rule in a collection of AB rules by concatenating the current states of each entity in the motif and generating a cohesive state. This cohesive state can then be used to perform a constant time lookup for the matching cohesive state in our collection of rules which will yield the appropriate action. This process is far more efficient than performing a linear evaluation of each condition for every rule, which is what is currently done for rules, as discussed in the literature review. The only downside to this is that we cannot use the elaborate mathematical conditions typically used in rules, such as those seen in [17]. However, these kinds of conditions are of little to no importance in the realm of policy enforcement as our main concern is on the facilitation of environmental state transfer, as seen in [1].

One thing to note is that this setup for rules precludes the possibility of conflicting rules (i.e., policy conflicts). This is because there is a one-to-one relationship between any key-value pair of cohesive states and actions in the rule. This relationship means that for any given motif, in its moment-to-moment operation, there is only one cohesive state that could possibly be applicable and therefore, only one action that could be executed. This is useful because we can simply take the different combinatorial variations of the cohesive state, and each one of them will be unique. Since these cohesive states are unique, we can use them to serve as a unique address for the rule and which can then be used to create a hash list for the rules. This reduces the time needed for rule selection to constant time because in order to select a particular rule; one simply needs to know its hash address, and the rule can be accessed in constant time.

Cohesive State Classification

In our development of the cohesive state concept, we have noticed four general classes of cohesive states in AB rules: 1) Necessary states, 2) Extraneous states, 3) Redundant states and 4) Error states. These classes of cohesive states arise from the various potential combinations of entity states within a given network motif. To illustrate these states, consider our ongoing smart door scenario from the beginning of this section. The user wishes for the door to emulate the following behavior: Every time the door is told to close by the user, it closes and locks itself, and every time the door is told to open by the user, it unlocks and opens. Given this scenario, the different classes of cohesive states are as follows:

Necessary States. A necessary state is any cohesive state that can be used to build the specific rules that the user wants. These states are non-error/non-redundant states that introduce useful interactivity to the users CPS. For example, a necessary cohesive state of our smart door may be “closed-locked, user-present, open”. In this case, we would want the CPS/IoT system to unlock and open the door if this cohesive state were to be encountered.

Extraneous States. An extraneous state is any cohesive state that is a non-error/non-redundant state that would be used to build rules that the user doesn’t want or need. An example of this would be the cohesive state “closed- locked, user-present, unlock”. In this case, as mentioned before the user only wants the door to unlock when it is told to open. Thus, this cohesive state is extraneous because it would be used to define behaviors that do not align with the user's wishes.

Redundant States. A redundant state is a cohesive state where the stimulus for action is equal to the current state of the given actuator. For instance, in the above example of a smart door, the cohesive state “open-unlocked, user-present, open” would be redundant because it asks the user what to do when the door is open, and the user tells the CPS/IoT platform to open the door. For these cohesive states, the answer is always “do nothing”. Thus, contributing no useful information to the CPS/IoT platform on how to govern its physical environment.

Error States. An error state occurs when the cohesive state contains individual entity states that spawn a cohesive state that we should never encounter in the environment. For instance, in the smart door example, the cohesive state “open-locked, user-present, close the door” would be an error state since the instance where the door is opened and locked should not occur.

These cohesive states can be used directly in the task of operational policy generation. Since there is a one-to-one relation between a necessary cohesive state and an action, the generation of rules can be simplified to the classification of the rules cohesive state and the selection of an action to be executed should the state be rated as necessary. However, because the user is the ultimate arbiter in whether a particular rule is satisfactory, and by extension, if a cohesive state is necessary, the collection of user input is vital to the process of operational policy generation as will be discussed in the next subsection.

User Evaluation of Cohesive states and their input in Operational Policy Generation

As mentioned earlier, human input takes a very important role in the generation of operational policy for CPS/IoT systems. For all CPS/IoT systems, during their initial setup, a person must define and approve of its core operational logic. For the purpose of operational policy generation in smart homes, we call this person the "user". With this in mind, any policy generation algorithm must, therefore, consider whether the user is satisfied with the rules generated by the algorithm, and by extension, the overall operational policy itself. To this end, we have identified three general options to gather human input for this problem:

1. Directly ask the user to rate the generated rules and use this input to build the policy.
2. Build a predictive model based on the user's behavior with the environment and use it to generate the policy autonomously.
3. Ask the user to write the policy themselves.

Each option revolves around user-provided input in the generation of operational policy in some fashion. Option 1 above is, in essence, using direct interaction with the user as the model for determining if a generated policy is satisfactory. Option 2 is a more autonomous method that would be more useful for individuals who do not want to or cannot provide the input necessary for the other methods. Finally, option 3, asking that the user write the policy themselves, is what is currently done across the fields of CPS/IoT, as discussed in the literature review. For the purpose of this thesis, we will be exploring operational policy generation algorithms that use option 1 as the primary way to gather human input. In future work, we may explore algorithms that utilize option 2.

Because we are directly asking for input from the user in the policy generation process, we must consider both the environment the operational policy is being applied to and what form this human input will take. These two things are inescapable. However, this input can be somewhat limited to a simple measurement of user satisfaction. Thus, as shown in the section on operational policy, we are using a variable that is dependent on user input, which allows the user to exert control of the rule generation process.

Considering direct human input, all possible methods of direct input largely amounts to asking the user if they are satisfied with a particular rule and in aggregate an operational policy. As such, we can use a simple ratings-based schema to register user satisfaction with particular rules. Thus, the general process for any Policy Generation Algorithm (PGA) using this input schema is as follows:

1. Generate a rule.
2. Ask the user if the rule is satisfactory.

We can measure the performance of the PGA, therefore, by two primary metrics: A) The number of times the user is asked for input and B) The overall complexity of the algorithm. The overall speed or complexity of the algorithm would have no real impact as the overwhelmingly largest factor impacting the usability of the policy generation process would be the total number of user interactions multiplied by the time spent waiting on user input. Thus, one of the fundamental things to consider when designing an algorithm for policy generation is to reduce the number of generated extraneous, redundant, and error cohesive states which, in turn, would reduce the number of times the user is asked to verify cohesive states and select actions (i.e., building rules). This would have the net effect of increasing the performance of a given PGA.

Since we are utilizing this method of asking the user for input to validate AB rules, there are two additional steps we can take to reduce the overall number of user interactions with the policy generation algorithm. First, since the process of generating the rules can be stated in two steps: A) Generate a cohesive state and B) Attach an action to that state, the overall process of asking for user input can be shortened from asking the user to evaluate each rule to asking the user to evaluate the cohesive state and then to select the most appropriate action from the list of that actuator's available actions. This can be used to build the rule since, as discussed in the previous subsection, a rule is composed of a cohesive state and an action. This presents a significant benefit in cases where there are many actions a device may take for any given cohesive state since the user will now only need to perform two actions, namely classify the cohesive state as necessary and then select an action, instead of many (i.e., evaluate each potential rule for a given cohesive state).

Second, the cohesive states that are rated necessary or extraneous by the user can be saved to eliminate the need to rerun the algorithm every time the user wants to make an adjustment to the behavior of the CPS. This is useful since the process of policy generation occurs both during the setup of the user's CPS/IoT system and potentially during runtime with any change in user preference.

Operational Policy Format using AB Rules

As mentioned in the section on operational policy, one of the core considerations we need to take in the operational policy generation process is the format used to store the rules. Since human preference changes over time and with regards to different social or environmental contexts, we need a way to save previously generated rules and cohesive states in an elastic and flexible manner so that the user can change them on demand and inject the new rules into their CPS, resulting in an update on its behavior. To answer this problem, we use the following two data structures: The Operational Policy Definition (OPD) and the Rule Base. The OPD is the set of all of the active rules for a given CPS. These rules can then be enforced by the CPS/IoT platform by performing a simple lookup in the array using the encountered cohesive state and the Device ID to fetch the action that is supposed to be taken for any applicable environmental scenario as defined by the user.

The OPD, as shown in figure 4 below, consists of a 3-dimensional associative array, where the first-dimension keys are the unique ID's of each local group of actuators in the network and the value is another associative array. The second-dimension associative array consists of the ID's of each actuator and an associative array of rules for that actuator. In the third-dimension, the keys are those cohesive states rated necessary, and the values are the actions to be taken by the actuator the array is listed under.

```
{ "Local Group ID" : {
        "Actuator ID" :   {
                           "AC-State,Linked-state,…" : action,
                           . . .
                          },
                          . . .
                      }
   …
}
```

Figure 4: Operational Policy Definition Format

Thus, in order to find out what action should be taken in any given moment for a specific actuator, the CPS/IoT platform just needs to know what local group the actuator can be found in, the actuators ID, and the cohesive state of that actuators network motif. Once these have been found, the platform just needs to perform a constant time lookup to select, evaluate, and trigger the rule all in one step. In addition to the OPD as defined above, we have designed a class for AB-Rules under the schema shown in figure 5 . We build the rules as an object that contains a weight allowing us to easily gauge the user's satisfaction with the rule itself. This rating is given to us when the user selects a rule to be added to their OPD. The weights of the other rules are then automatically set to 0. This would allow rule recommendation systems to easily aggregate the ratings for a set of rules and recommend rules to individuals over time shortcutting the need to rerun the policy generation process when the user changes their mind.

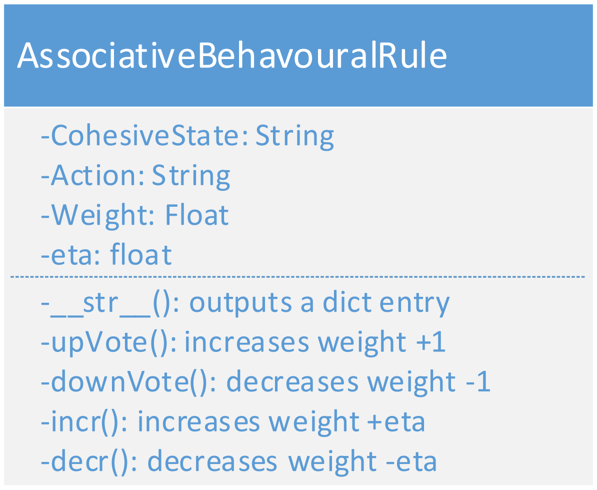
AssociativeBehavouralRule
-CohesiveState: String
-Action: String
-Weight: Float
-eta: float
-__str__(): outputs a dict entry
-upVote(): increases weight +1
-downVote(): decreases weight -1
-incr(): increases weight +eta
-decr(): decreases weight -eta

Figure 5: Associative Behavioral Rule class diagram

These rules are generated for the cohesive states rated necessary or extraneous and collected in a rule base, shown in figure 6 below, over the course of the policy generation cycle. The general format of the rule base consists of a similar format to the OPD, as seen in figure 4, with local groups holding a collection of actuators. The main difference is that each cohesive state in the rule base is a key to a collection of our AB-rule objects instead of actions as shown in figure 6 and with an example of the rule base shown in figure 7.

```
{ "Local Group ID" : {
"Actuator ID" : {
     cohesiveState1 : [rule1, rule2],
     cohesiveState2 : [rule3, rule4],
     cohesiveState3 : [rule5, rule6],
     cohesiveState4 : [rule7, rule8],
     . . .
}
. . .
     }
}
```

Figure 6: Rule base format

```
{ "MyHub" : {
"Smart-Door" : {
     "closed-locked, open the door": [AB-Rule(),AB-Rule()],
     "open-unlocked, close the door" : [AB-Rule(),AB-Rule()]
               }
     }
}
```

Figure 7: Rule base for the smart door example

This, in essence, gives us two sets of rules: an enforceable set of rules that the CPS/IoT platform uses (the OPD), and a base set that contains all of the generated rules including the ones in the OPD. This schema allows for a great deal of plasticity in the way the user manages their OPD since the user can simply select the rule they desire from the rule base and inject that rule into the OPD whenever they change their mind on how the CPS/IoT system should operate. The OPD can then be updated during runtime to alter the CPS/IoT systems behavior should the CPS/IoT platform allow rule injection.

OPERATIONAL POLICY GENERATION ALGORITHMS AND THEIR DESIGN

As mentioned in the introduction and literature review, one of the current problems in CPS/IoT research is the lack of algorithms that can generate the rules concerning the moment-to-moment operation of a smart home CPS/IoT system. In this section, we will be presenting three Policy Generation Algorithms (PGAs): a brute force algorithm, an algorithm adapted from the field of Evolutionary Computing, and a novel algorithm.

Brute Force Policy Generation

Typically, in order to evaluate the performance of any algorithm, the algorithm in question must be compared to a baseline. This baseline can be either be a well-known algorithm in the field or, as is our case where there are no well-known algorithms, a brute force approach to the problem. Thus, the first algorithm we will present is our baseline brute force policy generation algorithm that all other algorithms can be compared to. As its name suggests, the brute force algorithm accomplishes its central task by manually generating all possible cohesive states for a given NDF, then directly asking the user to evaluate each state and if the cohesive state is evaluated to be necessary, select a rule to be enforced.

The algorithm itself is shown in Algorithm 1 and is using the notation from the mathematical definition provided in the section on operational policy. In step 5, we first generate a matrix containing the current actuators states and the states of all connected entities within the current actuators network motif. From here, we can get the set of all potential cohesive states by computing the product of the matrix of states **M** with itself as shown in step 6.

```
//Given the set of entities E
1.  Rule Base ← ∅
2. OPD ← ∅
3. For each E_i in E:
4.    If(A is not empty):
5.        M ← S + I
6.        cohesiveStates ← M × M
7.        For each cs in cohesiveStates:
8.            cs.class ←  Ask the user to evaluate cs
9.            if(cs.class == "Necessary"):
10.               rules ← Generate all possible rules for cs
11.               rule ← ask the user to select a rule from rules
12.              Add rule to OPD and add rules to  Rule Base
13. Return OPD, Rule Base
```
Algorithm 1: Brute Force Policy Generation Algorithm

Next, in steps 7-12 of this algorithm, we then ask the user to evaluate the class of each cohesive state, as mentioned in the section detailing cohesive state classification. If the cohesive state is evaluated as being a necessary state, then the PGA will generate the rules associated with it and ask the user to select the rule they desire. This rule is then added to the active set of rules, the OPD, and all of the generated rules are archived in the rule base for ease of access should the user decide in the future to change the enforced rule.

This algorithm presents the least desirable method for generating the operational policy since it requires user interaction in the evaluation of all classes of cohesive state. As mentioned previously, the best way to increase the performance of the policy generation process is to reduce the overall number of user interactions with the PGA by limiting the number of cohesive states the user must classify. To this end, we are presenting two algorithms with the potential to yield better results: Differential Evolution, and A-Posteriori Policy generation.

Policy Generation via Evolutionary Computation

Evolutionary computation presents many algorithms which are useful in solving combinatorial optimization problems such as our policy generation problem. The general "goal" of these algorithms is to search through the "space" that all candidate solutions exist to find either the solution or solution set that will yield the greatest reward. These types of search algorithms can be used in our application domain to greater efficacy than our baseline brute force algorithm since in the process of searching for the optimal solution set the algorithm has a chance to bypass the evaluation of many candidate solutions in the search space. This presents a net benefit to our policy generation process since the worst case is to evaluate all solutions, which is what brute force does, and both the average and best cases involve generating the operational policy without evaluating all of the potential cohesive states.

One algorithm that is commonly applied to a variety of problems is differential evolution [35]. Differential evolution (DE) is a stochastic direct search and global optimization algorithm. In other words, it is an algorithm that searches for the optimal solution using an initially random solution set and does so by internally optimizing its solution set over successive generations. As such, Differential Evolution involves maintaining a population of candidate solutions that are subjected to successive iterations of recombination/mutation, evaluation, and selection in order to generate a final solution set that contains the most optimal candidate solutions. Depending on the application domain, the user can take either the best member of that set or the set itself. In Algorithm 2 shown below, we show the general algorithm for DE adapted from [36].

Differential evolution has been applied to combinatorial optimization problems with great success in the past. For instance, in [37], the authors apply DE to the NP-hard combinatorial optimization problem of complex network analysis. In [38, 39], the authors apply DE to a variety of hard combinatorial optimization problems such as feature subset selection and fuzzy-neuro adaptive systems.

Because of this, we have identified DE as an algorithm with the potential to be applied to our domain of operational policy generation. However, for the algorithm to truly be useful to our domain, several key processes must be adapted. Therefore, we present our adapted DE as shown in Algorithm 3 followed by a discussion of each of the core processes of DE, presented in Algorithm 2, and how they were adapted, as seen in Algorithm 3.

```
//Given $Population_{size}$, $Problem_{size}$, $Weighting_{factor}$, $Mutation_{rate}$
1. Population ← Generate Initial Population
2. $P_{fitness}$ ← Evaluate(Population)
3. Best ← Select most fit individual from Population using $P_{fitness}$
4. Until stop condition is met do:
5.     NextGeneration ← ∅
6.     For each individual $P_i$ in Population:
7.         $Parent_1 \leftarrow P_i$
8.         $Parent_2$ , $Parent_3$ ← $P_{random}$, where $P_{random}$ is unique for both parents
9.         Child ← initialize child
10.        For position i in each gene in $Parent_1$ with random CrossoverPoint:
11.            If based on $Mutation_{rate}$ it is time to mutate and i >= CrossoverPoint:
12.                $Child_i \leftarrow Parent_{3i} + Weighting_{factor} \times (Parent_{1i} - Parent_{2i})$
13.            Else:
14.                $Child_i \leftarrow Parent_{1i}$
15.        If the fitness of Child is greater than or equal to the fitness of $Parent_1$:
16.            NextGeneration ← Child
17.        Else:
18.            NextGeneration ← $Parent_1$
19.    Population ← NextGeneration
20.    $P_{fitness}$ ← Evaluate Population
21.    Best ← Select most fit individual from Population using $P_{fitness}$
22. Return Best
```

Algorithm 2: General differential evolution algorithm

```
1.  //Given Population_size, Weighting_factor, Mutation_rate, and the set of Entities in the CPS E
2.  Rule Base ← ∅
3.  OPD ← ∅
4.  For each E_i in E:
5.     If(A is not empty):
6.        M ← S + I
7.        Population ← Randomly generate Initial population of cohesive states using M
8.        Evaluate Population with user input
9.        Until user is satisfied, or number of genomes generated = |M × M | do:
10.          NextGeneration ← ∅
11.          For each individual P_i in Population:
12.             Parent_1 ← P_i
13.             Parent_2, Parent_3 ← P_random, where P_random is unique for both parents
14.             Child ← initialize child
15.             For position i in each gene in Parent_1 with random CrossoverPoint:
16.                If based on Mutation_rate it is time to mutate and i >= CrossoverPoint:
17.                   Index ← Parent_3i[1]+ Weighting_factor × (Parent_1i[1] − Parent_2i[1])
18.                   Child_i ← M[i][round(|Index|)]
19.                Else:
20.                   Child_i ← Parent_1i[0]
21.                Child.class ← Ask the user to evaluate cs
22.                if(Child.class == "Necessary"):
23.                   rules ← Generate all possible rules for cs
24.                   rule ← ask the user to select a rule from rules
25.                   Add rule to OPD and add rules to Rule Base
26.             If the fitness of Child is greater than the fitness of Parent_1:
27.                NextGeneration ← Child
28.             Else If the fitness of Child is equal to Parent_1:
29.                NextGeneration ← Child
30.                NextGeneration ← Parent_1
31.             Else:
32.                NextGeneration ← Parent_1
33.          Population ← NextGeneration
34. Return OPD, Rule Base
```

Algorithm 3: Adapted Differential evolution algorithm

Genomic Representation and Generation. The potential candidate solutions, called genomes or individuals, are the primary focus of the evolutionary algorithm. As the algorithm continues, its overarching goal is to create a population of the fittest individuals. As such, the "representation" or format of the genome is of paramount importance.

As mentioned in the section detailing user input, to generate a rule, we need two items: a cohesive state, and an action. For our adaptation of DE, the subject of evolution will be the cohesive state since every potential cohesive state can be classified according to the classifications discussed in the section detailing cohesive state classification and these classifications possess an evaluative quality (i.e., necessary states are more useful than error states).

Evolving the cohesive states is more efficient than evolving the rules themselves since for every cohesive state there may be many rules, and the evaluative quality (i.e., fitness) of a rule is largely dependent on its cohesive state. This allows us to bypass a significant amount of work for the user. Therefore, the cohesive state must be generated in a format that is useable by DE. To begin, as mentioned in the section detailing cohesive state classes, the default format of the cohesive state is as a single string consisting of comma delineated states for the given actuator and all of its linked entities as shown in (7) below:

$$< actuator\ state >, < LE_1 State >, < LE_2 State >, \ldots \tag{7}$$

However, as mentioned in [35], DE works with continuous real-valued functions using the weighted difference mutator, shown in line 12 of Algorithm 2, which won't work with our default comma delineated string. Therefore, we will need to reformat the cohesive state so that we can derive a real-valued function from the cohesive state.

In our adapted algorithm, Algorithm 3 lines 1-5, we are given the set of entities within the CPS: **E**, this superset contains the array of states for the entity, S, and may also contain an array of linked entities: I. By stacking these arrays together, we can create a matrix of states M that can be used to generate all potential cohesive state. There is a greater benefit to using this matrix of states, of course, and that is that we can use the indices of the elements in the matrix for our weighted difference mutator, discussed in more detail in the subsection on mutation, since the indices form a linear real-valued function in the range [0, n]. Thus, for our adaptation of DE, we use the genomic representation shown in (8):

$$[(acuator\ state, index), (LE_1\ state, index), (LE_2\ state, index), \ldots] \tag{8}$$

Where each tuples position in this form corresponds to the entities position in the matrix M (i.e., the row index), which is the same as in the traditional cohesive state format discussed in the section detailing the cohesive state's format. As seen in (8), each tuple consists of a state pulled from the matrix **M** and the column index of that state. The tuple format shown in (8) continues for each entity in the network motif for the given actuator (i.e., an actuator and all of its linked entities).

As an example, consider the following state matrix, seen in Table 1, that would be generated for the smart door from our ongoing example introduced in the section on operational policy. To generate a cohesive state from this matrix, we would begin with an empty array. Then for each entity in the matrix, we randomly choose a state from its array and save both the state and its index in a tuple that is appended to our genome. In (9), we see an example genome generated from this process.

Table 1: State Matrix for the smart door

Device	State 1	State 2	State 3	State 4	State 5
Door	open-locked	open-unlocked	closed-locked	closed-unlocked	N\A
Cam	user-present	no-user	unknown-person	known-person	N\A
Mic	open	close	lock	unlock	no-command

$$[(closed - locked, 3), (user - present, 1), (close, 1)] \tag{9}$$

This process of randomly generating the genome is what gives DE the ability to skip potential candidate solutions that are less fit, and it improves its ability to find good candidate solutions in the search space as discussed in depth in [35]. This potential to skip candidate solutions occurs because the algorithm itself is focused on finding cohesive states the user needs. Once this set of states is found, then the user can exit the algorithm without needing to rate all potential cohesive states. This factor also gives us our best case since there is a chance that all of the cohesive states the user wants may be generated during the initial stage.

The Process of Parental Selection. The next core process of DE concerns the selection of parents to breed the next generation. This process is done iteratively for each generation of candidate solutions and aids in increasing the diversity of the population while also constraining its evolution to candidate solutions who are more fit. Typically, in evolutionary computation, parents are selected based on their own fitness as a candidate solution using a variety of methods from round-robin selection to tournament selection.

However, as is the case with most DE algorithms such as [35-39], selection is done by randomly choosing three parents from the current population as shown in Algorithm 2 lines 7-8 and in lines 12-13 in Algorithm 3. While there are works that propose fitness-based methods of selection, such as in [40], for our particular problem, these other methods yield no tangible benefit as the fitness of a particular genome is directly determined by the user and said genomes consist of combinations of states.

Thus, even if good parents are chosen, the resulting child may still be rated worse than the parents as will be shown below. As such, a simple random selection of parents from the population as is done in [35-39] will suffice. To give an example, to be discussed in more detail in later, consider the following three parents rated "necessary" from our smart door example:

- $[(closed - locked, 2), (user - present, 0), (open, 0)]$
- $[(open - unlocked, 1), (no - user, 1), (no - command, 4)]$
- $[(open - unlocked, 1), (user - present, 0), (close, 1)]$

Using our adapted weighted difference mutation operator shown on line 17 of Algorithm 3 and a crossover point of 1 along with a weighting factor of 0.5, a child that would be rated "redundant" could be generated such as the example shown in (10) below. In this case, even though we have three good parents, the child that was generated from them is less fit then they are. This demonstrates the futility of using extra computational resources in the selection process for this problem.

$$[(closed - locked, 3), (user - present, 1), (close, 1)] \tag{10}$$

Mutation and Recombination. After selection is performed, a genetic operator is applied to the parents in an effort to generate a child that is similar in some respects to the parents yet different. This process is what allows the algorithm to search the available space of candidate solutions in an orderly fashion and is what principally separates genetic algorithms from Monte Carlo, i.e., random search, algorithms.

Mutation in DE is done a bit differently than in other genetic algorithms and is the main selling point of using DE. In DE, mutation is done in concert with recombination and uses the weighted difference operator shown in (11) where i is the index of a specific gene as shown in lines 10-14 of Algorithm 2.:

$$Parent3_i + Weightingfactor \times (Parent1_i - Parent2_i) \tag{11}$$

The weighting factor thus affects the amount a specific gene mutates. If this value is set too high, then the mutation will jump around instead of shift. On the other hand, if the weighting factor is set too low, then the likelihood of the gene staying the same as one of its parents increases greatly. The goal in this method is to increase diversity by way of small changes in genes over time, just like how biological evolution functions. Recombination occurs during the same process as the mutation rate determines when our mutation operator will be used. When it is not in use, the algorithm will simply use the parent gene, as shown in line 14 of Algorithm 2. This method of doing mutation and recombination is particularly effective when the fitness of candidate solutions is evaluated by objective functions that are nondifferentiable, non-continuous, non-linear, noisy, flat, multi-dimensional or have many local minima, constraints or stochasticity [35].

Since our problem is a selection problem at its core (we must select the right cohesive states), using the real-valued difference of two cohesive states for recombination and mutation would not work as discussed in the previous subsections. It also does not make any sense to pursue this avenue since the states may be of different data types and will not be a continuous real-valued function as is needed by DE. The indices of each state's position in the state matrix M, which are unique integer values that correspond to a particular state, on the other hand, are real-valued linear functions with the range [0, n]. As such, our mutation operator will function perfectly with them and will enable us to choose different states by mutating the index in a deterministic way using DE's real-valued difference mutator, as shown in Algorithm 3 lines 15-20. This would allow us to generate new cohesive states that are different yet similar to their parent states.

To show how this works, consider our on-going smart door example, see Algorithm 3 lines 15-20 for the pseudocode to this process. To start off with, let's use a crossover point of 1 along with a weighting factor of 0.5, and a mutation rate of 50% for demonstration purposes. Using the information contained in the set **E**, we can generate the state matrix **M** by attaching the doors states to the states of its linked entities, as shown in Table 1 earlier.

Next, we can select three parents at random using the processes described in the previous subsection on parental selection and genomic generation:

- $[(closed-locked, 2), (user-present, 0), (open, 0)]$
- $[(open-unlocked, 1), (no-user, 1), (no-command, 4)]$
- $[(open-unlocked, 1), (user-present, 0), (close, 1)]$

From here, we use our weighted difference mutator, shown in (11), to generate a child. When i is 1, we simply place the corresponding gene from the parent into the child, resulting in the genome shown in (12). When i is 2, if it is time to mutate the gene, according to the mutation rate, then we mutate. For this gene, let's say it was not time to mutate. In which case, we then take the gene of the parent and place it in the child just like the previous step, as shown in (13).

Finally, when i is 3, if it is time to mutate the gene, according to the mutation rate, let's say it was, then we use our weighted difference mutator to mutate the gene and place it in the child. This is done by first mutating using the indices using the formula shown in (11) resulting in (14):

$$[(closed - locked, 2), (\quad), (\quad)] \tag{12}$$

$$[(closed - locked, 2), (user - present, 0), (\quad)] \tag{13}$$

$$1 + 0.5 \ \times (0 - 4) = -1 \tag{14}$$

Then we would take the absolute value of the index generated and round it to the nearest integer yielding an index of 1. Next, we use this index to grab the corresponding state from M. Since i is 3, we will look at the third device's row in M, and since the index we generated is 4, we will take the fifth state. This leaves us with the child shown in (15):

$$[(closed - locked, 3), (user - present, 1), (close, 1)] \tag{15}$$

Fitness and Evaluation. The fitness of a genome is the primary metric that the evolutionary algorithm uses to measure the survivability of said genome in its environment. The more "fit" a genome is, the more likely it will survive and reproduce. This is one of the primary drives in natural selection and is, therefore, one of the core processes of evolutionary computation. In evolutionary computation, we utilize a concept known as the "fitness function". In the broadest sense, a "fitness function" has only one primary quality: it just needs to be able to output a value that corresponds to the survivability of a given genome [35]. Fitness functions are also often called “objective functions” or “goal functions” [41], because they are often used to express how close a candidate solution is to the goal of a given problem. Depending on the application domain, these functions can be any one of the following archetypes: Formulaic, Simulative, or Evaluative.

Fitness Derived from the Evaluation of Mathematical Formula. Functions that use a mathematical formula to derive the fitness of a candidate solution fall into the class of formulaic fitness functions. This is the stereotypical fitness function, as shown in [35, 38-40], that uses a mathematical formula to determine a numeric fitness value for a given candidate solution. For instance, given a goal of seeking $\min(x)$ for $f(x) = x^2$, the fitness function could simply be how close a candidate solution x is to 0, (e.g. the evaluation of f(x)).

Fitness Derived from Simulation. Sometimes, the application domain’s environment is too difficult to represent mathematically, for instance, in aeronautics or in population dynamics. In these instances, the fitness of a candidate solution can be reasonably derived from testing or simulation such as done in [42-43]. Thus, our second class of fitness functions is the simulative class. These functions assess the fitness of potential candidate solutions by running them through a simulation and then analyzing the solutions performance results. In these cases, how the candidate solution performs in simulation is the basis for its fitness.

Fitness Derived from User Input. Our final class of fitness functions is the evaluative class. This class of fitness functions is reserved for those applications where human input is needed to evaluate the fitness of a candidate solution. This is the main subject of Interactive Evolutionary Computation (IEC) [44] and has been an active field of study for almost 20 years. For these types of fitness functions, direct human input is needed since the evaluation of a candidate solution may be either too complex for formulaic representation or simulation, or the objective of the optimization algorithm may be subject to human desires or behavior [44]. In these cases, the best approach is to collect direct human input as the human is the ultimate arbiter in how fit a candidate solution may be. For instance, consider a scenario were an evolutionary algorithm is used to optimize a restaurant's menu for its customers were each candidate solution is a potential menu item. In this case, the only way to gauge the fitness of a candidate menu item would be to collect information regarding the menu item directly from the customer.

As discussed in the previous section on user input, we will be using direct human input as the fitness model for our rules. This is because the fitness of a particular candidate solution, a cohesive state, can only be determined by the user who designs the operational logic for the CPS/IoT system. After all, nobody is interested in a house that does what it wants without the homeowner's input or a power grid that makes its own decisions without input from the related authorities.

Thus, the rules generated by the adapted differential evolution algorithm must conform to the user's expectations for the operation of their CPS/IoT system. This fact places any potential fitness function strictly in the evaluative class. It is at this juncture, after a child has been generated, that we would directly ask the user to determine the child's class, as discussed in the section detailing the cohesive state, and, if the class is evaluated as necessary, select the rule the user desires as shown in lines 21-25 of Algorithm 3. For the purpose of our algorithm, a rating of "necessary" has the greatest fitness value attributed to it, while the others have an equally low fitness value. This is because the objective of the algorithm is to search for necessary cohesive states until either the user tells it to stop or the algorithm has searched through the entire search space.

A-Posteriori Policy Generation

The term A-Posteriori as an adverb is taken from Latin meaning “knowledge relating to or derived by reasoning from observed facts”. It is used to describe logical processes that are based entirely on previously known information in order to reach a particular conclusion and is the bedrock of deductive reasoning. This is in direct contrast to the term A-Priori, which concerns “knowledge relating to or derived by reasoning from self-evident propositions”; A fundamentally inductive task. For example, the statement "Every mother has had a child" is an a-priori statement, since it shows simple logical reasoning and isn't a statement of fact about a specific case. Whereas the statement "This woman is the mother of five children" would be an a-posteriori statement since the speaker could only know this from previous experience (e.g., asking the woman or looking at her records).

In operational policy generation, a-posteriori reasoning can be of significant use due to the frequency of commonly used network motifs, as discussed in the section detailing network motifs. For instance, consider the smart homes in a residential district; in this district, there will undoubtedly be an abundance of smart doors such as the ones mentioned throughout this thesis. Thus, the network motif: *Microphone, Cam → Door* would be so prevalent in the residential district that the task of generating the operational policy for the smart door could be significantly shortened by referring to the previously generated cohesive states for this kind of smart door.

Of course, there are two major considerations to be taken here: The first concerns how environmental or social context may alter the operational policy for a given CPS. This is because the people who live in the CPS/IoT system’s physical environment may disagree with the way another person has their operational policy configured. They may also abruptly change their own wishes for a particular configuration when given a particular social context, e.g., having guests over, or an environmental context, e.g., the front door versus a bedroom door. For instance, consider the scenario stated in the section on operational policy. For sally's home, she has her own ideas about how the CPS/IoT system should govern the physical environment in her home, which her neighbors may not share. Of course, she may also change her mind on how her environment should be managed during the summer versus the winter or when her friend comes over. Thus, cataloging the policy and its rules and recommending them to others would be an inefficient way of using a-posteriori logic. A better method would be to retain the classifications given to each cohesive state and then use the aggregation of those classes with fresh user input to generate the operational policy.

Following this logic, different users may rate the same cohesive state differently. In the section discussing the Cohesive State, we discussed four different classes of cohesive states: 1) Necessary states, 2) Extraneous states, 3) Redundant states, and 4) Error states. To summarize: A necessary state is any cohesive state that can be used to build the specific rules that the user wants. An extraneous state is any cohesive state that is both a non-error and non-redundant state that would be used to build rules that the user doesn't want. While both are valid states, the difference is whether the user will use them or not. A redundant state is a cohesive state where the stimulus for action is equal to the existing state of the given device. And finally, an error state occurs when the cohesive state contains individual device states that spawn a cohesive state that we should never encounter in the environment or would violate a device's operating constraints.

Given these considerations, the best way to use a-posteriori logic in operational policy generation is to collect information regarding any particular network motifs and the cohesive states they spawn into a database and use this information to build the operational policy. This database consists of the collection of tables shown below in figure 8 below and contains information regarding the network, (i.e., the set E, as mentioned in the section on operational policy). This information is provisioned by the user and given to the algorithm before it begins.

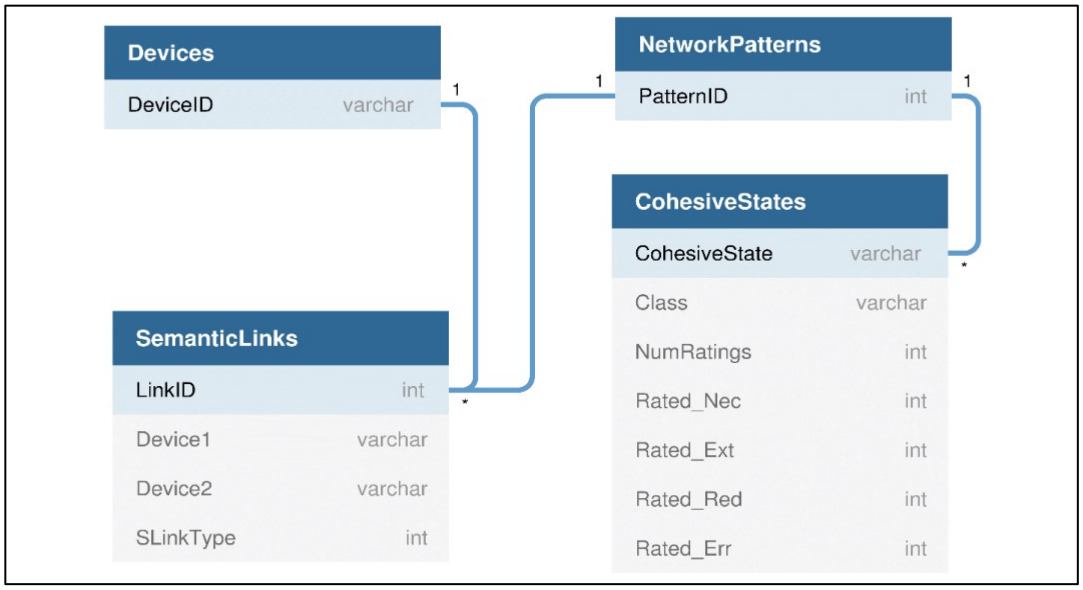

Figure 8: The Device Patterns Database Design

Given this database, we can then build our policy generation algorithm using a-posteriori logic, as shown below in Algorithm 4. The A-Posteriori PGA begins by finding the given actuators network motif, shown on line 6. As discussed previously, the network motif consists of the IDs for the actuator itself and the linked entities represented in the set I. With this information we can now find any previously generated cohesive states for this motif that are more likely to be classified by the user as necessary states. This is done by aggregating the individual class ratings given to us by the users for each cohesive state and selecting the class with the highest rating as the states "official" class. Thus, if a particular cohesive state is rated more times to be necessary, then it is more likely to be shown to the user first.

```
1. //Given the set of entities E, a Policy generation algorithm PGA, and a database
2. Rule Base ← ∅
3. OPD ← ∅
4. For each E_i in E:
5.    If(A is not empty):
6.       motif ← (I→ E_i)
7.       If(motif in database):
8.          classes ← [Rated_Nec, Rated_Ext, Rated_Red, Rated_Err]
9.          For class in classes:
10.            cohesiveStates ← query database for cs where class is max
11.            For each cs in cohesiveStates:
12.               rating ← ask user to rate cs
13.               If(rating == "necessary"):
14.                  rules ← Generate all possible rules for cs
15.                  rule ← ask the user to select a rule from rules
16                   Update the cohesive states rating in database
17.                  Add rule to OPD and add rules to  Rule Base
18.            Ask the user if they wish to continue else quit
19.      Else:
20.         cohesiveStates, ratings ← PGA(ACU)
21.         INSERT motif, cohesiveStates, ratings into database
22. Return OPD, Rule Base
```

Algorithm 4: The A-Posteriori Policy Generation Algorithm

This is useful to us since the cohesive state would be subject to less volatile differentiation than the rules themselves. For instance, consider our ongoing smart door example. In this scenario, the cohesive state: *[closed-locked, user-present, open]* is much more likely to be consistently rated necessary than the rules that could be generated from this cohesive state as different users may wish for the door to behave differently when given this scenario. However, since users may have differing opinions as to the rating of a particular cohesive state, they will still need to rate it themselves as shown in lines 10-17.

This variance in opinion would often occur when there are non-listed social or environmental contexts that affect the classification of cohesive states. For instance, consider the difference between two doors in a family home: a parent's bedroom door and a child's bedroom door. In some families, the parents may set a rule where the children cannot lock their door except in the case of an emergency, whereas other families may not have any such rule. In such instances, the difference between the parent's door, which can lock on command, and the child's door, which can only lock on emergencies, is simply that the door is to the child's bedroom and that this family has such a rule. This difference in social context would wildly alter the classification of the doors cohesive states even when given the exact same network motif.

If, after viewing all necessary states the user still has not viewed the cohesive states that they need, then we proceed to show the user the other classes of cohesive states in descending order. This descending order also considers the ratings of the cohesive states themselves. Thus, extraneous states that have more necessary ratings are higher on the list to show the user and those with less necessary ratings are lower. can go to the cohesive states that are more likely to be extraneous. If at any point in the process, the user has found all of the states they need and has selected the rules they want, they can exit the algorithm.

Of course, if the particular network motif has not been seen before, then the a-posteriori algorithm will need to use a sub-PGA to generate the policy as shown on lines 19-21 in Algorithm 4. This can be any algorithm suited for generating operational policy from scratch, such as the brute force algorithm or the differential evolution algorithm. This step is where the database gets its initial information regarding the network itself. On line 6, we get the network motif, so all that is left to collect is the cohesive states and their ratings. Of course, the user does not need to run the algorithm twice as the sub-PGA will also update the OPD and the rule base.

One thing to note about the a-posteriori algorithm is that it is not a recommender system under the commonly used definition [11, 12]. While both a-posteriori and recommender systems make use of user ratings, a recommender system attempts to predict future user preferences and recommend new items to the user. This is in contrast to a-posteriori, which uses the ratings to order the search for items the user may want and then uses the selected items from the user to build a final product. The key difference between recommender systems and a-posteriori is the use of prediction in order to recommend new items and services to the user. To give an example in our domain of policy generation, a recommender system would utilize techniques such as collaborative filtering to predict which rules the user is most likely to rate as necessary. In contrast, a-posteriori uses previous user ratings to order a direct search for necessary states and makes no effort at predicting whether the states are more likely necessary.

EXPERIMENTAL SETUP AND RESULTS

In this section, we will discuss the setup and results of an experiment concerning the generation of operational policy and then provide analysis of the results along with a comparison of the performance results for each algorithm. The experiment itself consists of running each of our three policy generation algorithms (Brute Force, Differential Evolution, and A-Posteriori) on a collection of network motifs of increasing size using a deterministic model for user input regarding cohesive state classification and associative-behavioral rule selection. This experiment allows us to collect data regarding the performance of the algorithms on a variety of motif shapes and sizes.

Experimental Setup

For this experiment, we will be running the policy generation algorithms over a collection of network motifs. As discussed in the section detailing network motifs, a network motif is a discrete sub-graph of a given network. In our case, it consists of a primary actuator and its linked entities. For our experiment, the network motifs grow in size along two primary axes: the number of linked entities and the number of states per entity. Therefore, the growth of the entire operational policy space that any potential cohesive state and its associated rules belong to can be given by the relationship shown in (16) , where D is the primary actuator in the motif and L is the collection of linked entities.

Thus, if we consider a primary actuator with 4 states and 3 linked entities that each have 2, 4, and 6 states respectively then the total number of cohesive states would be as shown in (17) below. Compare the number shown in (17) to a scenario where we have 20 devices with the same pattern of state numbers as shown in (18) below where we get over 10 septillion different states.

$$|\boldsymbol{D}_{states}| \times \prod_{i=0}^{|\boldsymbol{L}|} |\boldsymbol{L}_{i,states}| \tag{16}$$

$$4 \times (2 \times 4 \times 6) = 192 \tag{17}$$

$$4 \times \prod_{i=1}^{20} i \times 2 = 10{,}204{,}330{,}624{,}503{,}313{,}858{,}560{,}000 \tag{18}$$

This growth rate is worse than exponential growth, with the total number of cohesive states exploding to intractable levels as the number of devices and states grows. If we constrain the growth of the motifs to where the number of states is equal to the number of devices in the motif, then we have a growth rate of n^n, which demonstrates just how unreasonable the size of the policy space grows.

While the algorithms themselves can handle this growth rate, if given the time and extensive computing resources, using a super-massive environment for our simulation would be unnecessary since we would have a good idea of the differences between the algorithms long before they finished. With this consideration in mind for our simulation, we constrained the number of linked devices to be 8 and the maximum number of states per device to be 6. The growth rate for this particular constraint is shown in figure 9. This is a reasonable constraint as the domain of our work concerns smart homes. For example, consider our ongoing smart door scenario. While the home owner may have numerous devices in their home, the smart door itself does not need to be aware of their existence as it is only concerned with its own motif.

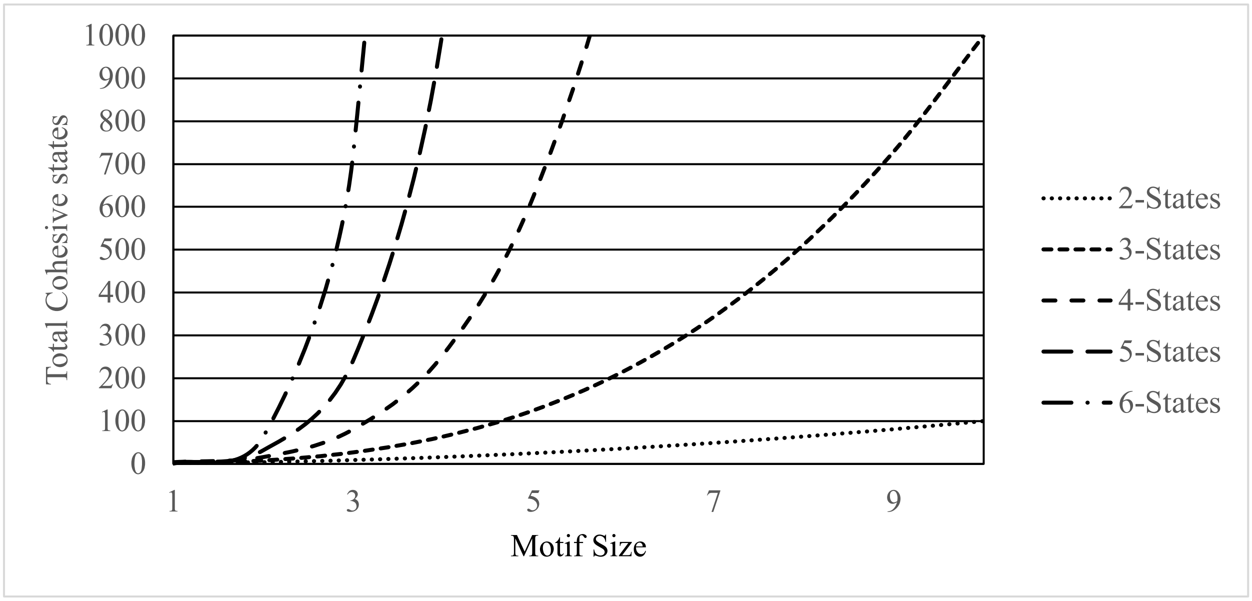

Figure 9: Growth of total cohesive states in motifs with states ranging from 2-6.

To simulate network motifs with this growth rate and examine how each algorithm performs, we created a file containing a dictionary of entities where each actuator and its linked entities (i.e., the network motif) are stored under a single local group. For instance, for the motif of size 2-2 (i.e., two devices with two states each) there is one local group and so on for the rest of the motif sizes. For reference purposes, we are calling this file the Network Definition File (NDF). From here, the actuator is then generated according to figure 10.

```
actuator = {
"Entity-ID" : "PA-"+str(size[0])+str(size[1]),
"States" : [str(x) for x in range(size[1])],
"Actions" : [str(x) for x in range(size[1])],
"Linked entities" : []
}
```

Figure 10: The data format for primary actuators

Next, the linked devices are created according to figure 11. After the devices for a particular motif are created, we add the ID for each linked entity in the motif to the primary

```
linked_entity = {
"Entity-ID" : "LE-"+str(size[0])+str(size[1]),
"States" : [str(x) for x in range(size[1])],
"Linked entities" : []
}
```

Figure 11: The data format for linked entities.

actuators “linked entities” list, and the devices are added to the NDF under a local group named after the motif size. So, for a motif of size 2, entities with 2 states per entity (i.e., size 2-2), one primary actuator and one linked entity would be created and added to a local group named “LG-22”. This process is then repeated until we have a network with one motif per local group for every motif size in our simulation, as shown below in figure 12.

```
{ “LG-22” : {
                "PA-22" : {
                        "Entity-ID" : "PA-22”,
                        "States" : [“0”, 1”],
                        "Actions" : [“0”, “1”],
                        "Linked entities" : [“LG-22:LE-220”]
                            },
                 “LE-220” : {
                        "Entity-ID" : "LE-220”,
                        "States" : [“0”, “1”],
                        "Linked entities" : []
                             }
              },
…
}
```

Figure 12: An abbreviated example of the network definition used in this experiment

This process creates a network of 60 devices clustered into 12 local groups. Given such a network, we can now easily test how each policy generation algorithm performs with different sized motifs. One thing to note, however, is that this specific configuration was done to benchmark each algorithm’s performance on a variety of motif sizes. While we could use a real-world environment for testing, since the major focus of this experiment is to gauge the performance of these additional algorithms, A-posteriori and our adaptation of Differential Evolution, a simulated environment will suffice. This is because simulation will still allow us to answer the important questions regarding our performance criteria which were touched on in the section on user input and will be expanded on in the subsection on performance criteria below.

A Deterministic Model for Cohesive State Classification and Rule Selection

As mentioned in the previous sub-section, the size of the operational policy space in our simulation is quite large. At its largest, there are over 65,000 potential cohesive states for the user to classify. In an effort to mitigate the amount of work to be done in the benchmark, we have designed a deterministic model for user input. This model starts with the following specifications: First, the states are saved as a list of integers, in ascending order, within the range [0, size], where “size” is the upper limit of the number of states a device may have. For instance, for a motif with 2 entities where each entity has 6 states (i.e., size 2-6) the list of states would be: [0, 1, 2, 3, 4, 5]. As discussed in the section on cohesive state classification, we have defined four potential classes of cohesive states: A) Necessary states, B) Extraneous states, C) Redundant states, and D) Error states. One of the two core tasks of the user is to classify a given cohesive state. Thus, the second specification of our model is that there is an integer representation for each class:

- Necessary states are class 1
- Extraneous states are class 2
- Redundant states are class 3
- Error states are class 4

Ergo, if a cohesive state has a class of 1, then it is necessary. This sets up a useful situation since we have specified the states for every device to be integers. Thus, we can now apply the formula shown in (19) to each cohesive state in order to automatically obtain its class when given a cohesive state of the form: “<int>, . . . , <int>” :

$$class = \left(\sum states\right) \% \, 4 + 1 \tag{19}$$

This gives us a relatively even distribution, roughly 25% for each class, of the defined classes for the cohesive states. For instance, when given a motif of size 4-4, our algorithms may come across the following cohesive state: “0, 3, 2, 4”. To apply this model to get the state's class we would first apply the formula shown in (19) to the state; then after performing the calculation, we would get the result shown in (20).

$$(0 + 3 + 2 + 4) \% \, 4 + 1 = 2 \tag{20}$$

The value 2 corresponds to the class “Extraneous”, and we would only add 1 to the total number of user interactions since the class is not “necessary”, if it where we would add 2 to account for the selection of a rule. This model allows us to test each algorithm in a similar manner as the same model will be used in all three algorithms to simulate user input for the jobs of cohesive state classification and rule selection. This will model will allow us to track the number of user interactions needed to converge on a desired OPD for a given network motif. As stated previously, this number forms our core performance metric and will be discussed in more detail in the next subsection.

Performance Criteria

Of course, for us to be able to evaluate the performance of the proposed algorithms for operational policy generation, we must have a standard performance criterion to judge the algorithms by. While there are many factors one would initially think to use, factors such as speed or computational complexity offer no real useful information since the algorithm will be spending most of its time waiting for the user to classify a given cohesive state and select their desired rule. Other factors also run the risk of yielding biased results since we would be comparing an evolutionary algorithm against non-heuristic algorithms. In this case, the performance criteria must be something that can be used to judge all three algorithms on a level playing field without regard for their implementation details which would play a huge factor in the collection of something like a timing profile.

As mentioned in [44], a better method of measuring the performance of interactive algorithms lies in measuring the rate at which the algorithm converges on the users desired outcome. In the case of our policy generation algorithms, as mentioned in the section on operational policy, we can measure the rate at which an algorithm converges upon the user's desired operational policy by the following metrics: (A) the number of times the user was asked for input, and (B) the number of cohesive states skipped by the algorithm. This metric is fair to all three algorithms since it does not regard implementation details. Given this metric, we can calculate a number of other metrics to be used as well.

First, we can calculate the error of any particular algorithm by subtracting the number of necessary cohesive states for a particular network motif from the total number of user interactions observed. This is given by the formula shown in (21). In our case, the necessary user interactions are exactly equal to the number of necessary cohesive states plus the number of times we selected a rule, and since we only select a rule when a state is rated necessary, we can use the formula shown in (22).

$$Error = Total\ User\ Interactions\ - Necessary\ User\ Interactions \tag{21}$$

$$E = TUI - (2 \times Necessary\ CS) \tag{22}$$

This is because the other classes of cohesive states are all unnecessary for the operation of a given CPS/IoT system and contribute no useful information. Thus, it would be better if the user was not even asked to evaluate these cohesive states in the first place. Thus, we can say that the primary goal of any good policy generation algorithm is to produce an operational policy that the user agrees with while minimizing the error generated from the process (i.e., the number of times the user was asked unnecessary questions).

Experimental Results

In this section, we provide the experimental results from running all three algorithms (Brute force, Differential Evolution, and A-Posteriori) on the same network definition described at the beginning of this section, with the deterministic model described earlier on used to simulate human input. Since some of the algorithms are stochastic processes, the simulation was run 10 times, and the results of each run averaged and presented in tables 2-6 below. First, some information about the tables to make them easier to understand:

- Size – This is the size of the network motif.
- TUI – This is the total number of user interactions.
- TCS – This is the total number of cohesive states for this motif.
- TRated – This is the total number of ratings the model performed for this motif.

- Nec – This is the total number of necessary states
- Other – This is the total number of errors as discussed earlier
- Ext – This is the total number of states rated extraneous
- Red – this is the total number of states rated redundant
- Err – this is the total number of states rated error

As shown in Table 2, the brute force algorithm performs exactly as one would expect, given that it simply generates every possible solution to the problem. This means that the algorithm only converges on the desired operational policy after the user views every possible cohesive state. This provides the baseline that our other algorithms can be measured against.

Analysis of differential evolution performance results

For our simulation, DE generally performed better, as shown in Table 3, then brute force exhaustively searching for the collection of correct solutions. There were a few instances where DE resulted in exhaustively searching for the correct solutions. These mainly occur in smaller network motif sizes such as (2-2) or (4-2). As can be seen from a look at Tables 2 and 3, DE and brute force have similar performance when converging to the users desired answer on small scale sets. This leads to the conclusion that for small search spaces the use of the exhaustive Brute force Algorithm would be better than DE as the user will end up classifying every possible cohesive state anyways and doing so in an orderly fashion vs. DE's random fashion would better serve the user.

Where DE performs better than brute force searching is in regard to operational policy generation for medium-to-large scale network motifs. Here the random aspect of the algorithm gives it a chance to skip the non-necessary cohesive states by generating initially random populations of cohesive states and then performing a deterministic search throughout the operational policy space.

Table 2: Brute Force Policy Generation Results

Size	TUI	TCS	TRated	Nec.	Other	Ext.	Red.	Err.
2,2	5	4	4	1	3	2	1	0
2,3	11	9	9	2	7	2	3	2
2,4	20	16	16	4	12	4	4	4
4,2	18	16	16	2	14	4	6	4
4,3	102	81	81	21	60	20	20	20
4,4	320	256	256	64	192	64	64	64
6,2	80	64	64	16	48	12	16	20
6,3	911	729	729	182	547	182	183	182
6,4	5120	4096	4096	1024	3072	1024	1024	1024
8,2	328	256	256	72	184	64	56	64
8,3	8202	6561	6561	1641	4920	1640	1640	1640
8,4	81920	65536	65536	16384	49152	16384	16384	16384

Table 3:Differential Evolution Based Policy Generation Results

Size	TUI	TCS	TRated	Nec.	Other	Ext.	Red.	Err.
2,2	5	4	4	1	3	2	1	0
2,3	11	9	9	2	7	2	3	2
2,4	17	16	14	3	11	3	4	4
4,2	18	16	16	2	14	4	6	4
4,3	102	81	81	21	60	20	20	20
4,4	100	256	80	20	60	20	22	18
6,2	42	64	32	10	22	6	6	10
6,3	911	729	729	182	547	182	183	182
6,4	5120	4096	4096	1024	3072	1024	1024	1024
8,2	175	256	138	37	101	39	31	31
8,3	8199	6561	6558	1641	4917	1640	1638	1639
8,4	58311	65536	47051	11260	35791	11878	11914	11999

Table 4: A-Posteriori Policy Generation Results – With unknown network motif

Size	TUI	TCS	TRated	Nec.	Other	Ext.	Red.	Err.
2,2	5	4	4	1	3	2	1	0
2,3	11	9	9	2	7	2	3	2
2,4	20	16	16	4	12	4	4	4
4,2	18	16	16	2	14	4	6	4
4,3	102	81	81	21	60	20	20	20
4,4	320	256	256	64	192	64	64	64
6,2	80	64	64	16	48	12	16	20
6,3	911	729	729	182	547	182	183	182
6,4	5120	4096	4096	1024	3072	1024	1024	1024
8,2	328	256	256	72	184	64	56	64
8,3	8202	6561	6561	1641	4920	1640	1640	1640
8,4	81920	65536	65536	16384	49152	16384	16384	16384

Table 5: A-Posteriori Policy Generation - Known motif with ambiguous necessary states

Size	TUI	TCS	TRated	Nec.	Other	Ext.	Red.	Err.
2,2	4	4	3	1	2	2	0	0
2,3	6	9	4	2	2	2	0	0
2,4	14	16	8	6	2	2	0	0
4,2	9	16	6	3	3	3	0	0
4,3	71	81	41	30	11	11	0	0
4,4	207	256	128	79	49	49	0	0
6,2	47	64	28	19	9	9	0	0
6,3	605	729	364	241	123	123	0	0
6,4	3373	4096	2048	1325	723	723	0	0
8,2	228	256	136	92	44	44	0	0
8,3	5413	6561	3281	2132	1149	1149	0	0
8,4	53988	65536	32768	21220	11548	11548	0	0

Table 6: A-Posteriori Policy Generation - With known network motif

Size	TUI	TCS	TRated	Nec.	Other	Ext.	Red.	Err.
2,2	2	4	1	1	0	0	0	0
2,3	4	9	2	2	0	0	0	0
2,4	8	16	4	4	0	0	0	0
4,2	4	16	2	2	0	0	0	0
4,3	42	81	21	21	0	0	0	0
4,4	128	256	64	64	0	0	0	0
6,2	32	64	16	16	0	0	0	0
6,3	364	729	182	182	0	0	0	0
6,4	2048	4096	1024	1024	0	0	0	0
8,2	144	256	72	72	0	0	0	0
8,3	3282	6561	1641	1641	0	0	0	0
8,4	32768	65536	16384	16384	0	0	0	0

For these motif sizes, if the initial population size is set to the number of cohesive states, and therefore rules since they share a one-to-one relationship, that the user wishes to generate then there is a random chance that the algorithm will generate either a portion of these cohesive states or all of them in the initial population. This provides a chance for the user to skip all of the other types of states which cuts the number of total user interactions (TUI) as shown in Table 3, for motif sizes (4-4) and larger. In this case, the user would encounter a portion of the necessary cohesive states and then perform a limited search for the other necessary states that may be present in the search space with an overall lower number of user interactions.

However, there are two notable issues with using this algorithm to solve our problem. The first is the amount of reputation in generating solutions. As DE continuously generates solutions and shifts the gene makeup of parent solutions to create child solutions, there are a lot of candidate solutions that have been seen by the user before. This leads to the algorithm essentially spinning its wheels while the user waits on the next generated solution. And since there is no benefit to showing the user the cohesive states they have already classified; the user must wait until the algorithm generates a cohesive state the user has not seen before.

The second issue is also related to the first yet more significant. This concerns how the algorithm finds all of the cohesive states the user needs. It may be the case that the user has found all but a few cohesive states to run their CPS/IoT system as desired, and yet finding those last few leads to an exhaustive search. This leads to the worst-case performance that is worse than brute force searching for the last few cohesive states. This happens because the user will both need to exhaustively search for the last few cohesive states while also waiting through an increasingly larger amount of repetitive cohesive states that can drastically increase the wait times and overall user involvement in the policy generation process. These issues are the core reasons why some of the results for larger-scale sets in Table 3 are roughly equal to the results in Table 2.

Analysis of A-Posteriori performance results

As shown in Table 6, when given a motif the algorithm has seen before, AP automatically gives the user the cohesive states that are the more likely to be rated necessary. This significantly reduces the amount of interaction the user needs to take in all cases and forms our best case. For our particular experiment, we tested AP using its best, average, and worst-case scenarios. When the number of cohesive states rated necessary was equal to the number of cohesive states that we needed, this forms the best-case scenario for AP. However, this will not always be the case in the real-world. As many people use the algorithm, a certain amount of ambiguity comes into play in the classification of cohesive states. This arises because of the potential differences in user opinions as to how a particular network motif should behave. For some users a particular cohesive state that others rate as necessary may for them be extraneous. The same is true in reverse, as some states rated extraneous by the majority of users may be useful to a select few. This ambiguity in the ratings of cohesive states forms the average case whose results are shown in Table 5. In this case, the user will need to evaluate both the necessary states and the extraneous states. Resulting in some error as discussed earlier in this section.

This particular issue causes us to be a bit more nuanced in our evaluation of AP. However, in both the best and average cases, AP still performs much better than both Brute Force and DE when given a motif it has seen before. However, as can be seen in Table 4, when given a motif that the algorithm has not seen before AP must rely on a sub-process that can deal with generating policy from scratch. In our particular experiment, we deferred this process to brute force. This results in the worst case for AP, as shown in Table 4. However, AP does not mandate that we specifically need to use brute force for the sub-process. We could just as well substitute DE or any other algorithm that could generate policy from scratch.

CONCLUSION

In this thesis, we explored the concept of the “operational policy”, a plan for the operation of a CPS/IoT system, in smart homes and framed the task of building the operational policy as a combinatorial optimization problem that algorithms can solve. We then introduced the “associative-behavioral” rule, which constrains human input in the rule building process to selecting the actions they want their devices to take in a given scenario. From here, we introduced three algorithms for the generation of operational policy: a brute force algorithm, an adaptation of the differential evolution algorithm, and the a-posteriori algorithm. By comparison to what is commonly used in the scientific field today, as discussed in the introduction and the literature review, the use of algorithms to generate operational policy presents a significant benefit to users. This takes place in the form of not having to write the rules for their own CPS/IoT system by hand; a task that generally users spend much time on. By narrowing down the amount of work to classifying the conditions for a rule to trigger and then selecting an action; users are free to spend more time on designing the core operational logic for their smart home system instead of writing rules by hand. This serves to increase the overall usability of CPS/IoT systems in smart homes as policy conflicts are excluded due to the nature of AB rules, as discussed in the section on AB rules, and the core task of selecting the rules for necessary environmental states is easier than writing code or rules.

As for the algorithms used in our experiment, a comparison between them is largely dependent on the specific configuration of the smart homes’ physical environment after the user builds it. If the environment contains many network motifs that have not been seen by AP before, DE is the better choice as it performs better than brute force in the average case for the larger network motifs seen in our simulation. For smaller network motifs of the same quality, brute force is better because it has the same number of user interactions as DE without any of the extra processes, see Tables 2-3, while not having the issue of repetition. If the CPS/IoT systems physical environment contains motifs seen by AP before then in both the best and average cases, AP outperforms both brute force and differential evolution for both large and small network motif sizes as can be seen in Tables 5-6. This is particularly useful for CPS/IoT systems as the frequent use of common motifs in the real world (e.g., lights in smart homes or switches in smart grids) allows AP to shortcut the process of cohesive state generation by allowing it to recommend previously generated cohesive states.

Overall, all three algorithms present an increase in usability for CPS/IoT systems as they take what was previously an arduous task of writing the operational logic by hand and turn it into a selection task where the user just needs to select the appropriate action for any necessary environmental scenarios.

This research lays the groundwork for future research in the generation of operational policy. This work could be expanded upon with the introduction of new algorithms for policy generation as well as graphical interfaces that seek to increase the usability of the underlying algorithms by altering the way the user interacts with them. Further research could also be done on algorithms that conduct a semi-ordered search for cohesive states vs. doing a completely random search, as done in DE. This would potentially lead to getting the best of both worlds since in a random search we have the potential to skip error states, and in an ordered search we eliminate the possibility of generating previously seen cohesive states. This would be a great benefit to any future algorithms that do not rely on previous information in the generation of operational policy, such as our adaptation of DE.

REFERENCES

[1] C. Greer, M. Burns, D. Wollman and E. Griffor, "Cyber-Physical Systems and Internet of Things," National Institute for Standards and Technology (NIST), Gaithersburg, Maryland, Special Publication (NIST SP) - 1900-202, Mar. 7, 2019. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1900-202.pdf

[2] Y. Tan, M. C. Vuran and S. Goddard, "Spatio-Temporal Event Model for Cyber-Physical Systems," in *29th IEEE International Conference on Distributed Computing Systems Workshops*, Montreal, QC, 2009, pp. 44-50. doi: 10.1109/ICDCSW.2009.82

[3] R. Baheti and H. Gill, "Cyber-physical systems," *The Impact of Control Technology* (2011): 161-166.

[4] S. K. Datta, C. Bonnet and N. Nikaein, "An IoT gateway centric architecture to provide novel M2M services," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, 2014, pp. 514-519. doi: 10.1109/WF-IoT.2014.6803221

[5] S. K. Datta and C. Bonnet, "Smart M2M Gateway Based Architecture for M2M Device and Endpoint Management," in *2014 IEEE International Conference on Internet of Things (iThings)*, and *IEEE Green Computing and Communications (GreenCom)* and *IEEE Cyber, Physical and Social Computing (CPSCom)*, Taipei, 2014, pp. 61-68. doi: 10.1109/iThings.2014.18

[6] Y. Lee and S. Nair, "A Smart Gateway Framework for IOT Services," *2016 IEEE International Conference on Internet of Things (iThings)* and *IEEE Green Computing and Communications (GreenCom)* and *IEEE Cyber, Physical and Social Computing (CPSCom)* and *IEEE Smart Data (SmartData)*, Chengdu, 2016, pp. 107-114. doi: 10.1109/iThings-GreenCom-CPSCom-SmartData.2016.44

[7] J. Herwan, S. Kano, R. Oleg, H. Sawada and N. Kasashima, "Cyber-physical system architecture for machining production line," *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, St. Petersburg, 2018, pp. 387-391. doi: 10.1109/ICPHYS.2018.8387689

[8] Ogata, K. (2016). *Modern control engineering*. 5th ed. Pearson.

[9] A. Floris and L. Atzori, "Managing the quality of experience in the multimedia Internet of Things: A layered-based approach", *Sensors*, vol. 16, no. 12, pp. 2057, 2016.

[10] C. Sarkar, S. N. A. U. Nambi, R. V. Prasad and A. Rahim, "A scalable distributed architecture towards unifying IoT applications," *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, 2014, pp. 508-513. doi: 10.1109/WF-IoT.2014.6803220

[11] J. Bobadilla, F. Ortega, A. Hernando, A. Gutiérrez, "Recommender systems survey", *Knowledge-Based Systems*, Volume 46, 2013, Pages 109-132, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2013.03.012.

[12] P. Melville and V. Sindhwani, *Recommender Systems*, *Encyclopedia of Machine Learning*, 2010.

[13] A. Forestiero, "Multi-Agent Recommendation System in Internet of Things," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Madrid, 2017, pp. 772-775. doi: 10.1109/CCGRID.2017.123

[14] Y. Zhang, "GroRec: A Group-Centric Intelligent Recommender System Integrating Social, Mobile and Big Data Technologies," in *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 786-795, 1 Sept.-Oct. 2016. doi: 10.1109/TSC.2016.2592520

[15] S. Gao et al., "A Cross-Domain Recommendation Model for Cyber-Physical Systems," in *IEEE Transactions on Emerging Topics in Computing*, vol. 1, no. 2, pp. 384-393, Dec. 2013. doi: 10.1109/TETC.2013.2274044

[16] J. Shi, J. Wan, H. Yan, H. Suo, "A survey of cyber-physical systems", *Proc. Int. Conf. Wireless Commun. Signal Process. (WCSP)*, pp. 1-6, Nov. 2011.

[17] Cano, G. Delaval, and E. Rutten, "Coordination of ECA Rules by Verification and Control," *Lecture Notes in Computer Science Coordination Models and Languages*, vol. 8459, pp. 33–48, 2014.

[18] J. Cano, E. Rutten, G. Delaval, Y. Benazzouz and L. Gurgen, "ECA Rules for IoT Environment: A Case Study in Safe Design," *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, London, 2014, pp. 116-121. doi: 10.1109/SASOW.2014.32

[19] A. Paschke, "ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics", *International Conference of Rule Markup Languages (RuleML'06)*, pp. 1-30, 2006.

[20] C. Sarkar, S. N. A. U. Nambi, R. V. Prasad and A. Rahim, "A scalable distributed architecture towards unifying IoT applications," *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, 2014, pp. 508-513.
doi: 10.1109/WF-IoT.2014.6803220

[21] T. Perumal, M. N. Sulaiman, N. Mustapha, A. Shahi and R. Thinaharan, "Proactive architecture for Internet of Things (IoTs) management in smart homes," *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, Tokyo, 2014, pp. 16-17. doi: 10.1109/GCCE.2014.7031347

[22] E. G. Allan Jr., W. H. Turkett and E. W. Fulp, "Using Network Motifs to Identify Application Protocols," *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*, Honolulu, HI, 2009, pp. 1-7. doi: 10.1109/GLOCOM.2009.5425781

[23] Jiehan Zhou *et al.*, "CloudThings: A common architecture for integrating the Internet of Things with Cloud Computing," *Proceedings of the 2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, Whistler, BC, 2013, pp. 651-657. doi: 10.1109/CSCWD.2013.6581037

[24] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," in *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347-2376, Fourthquarter 2015. doi: 10.1109/COMST.2015.2444095

[25] M. Jbair, B. Ahmad, M. H. Ahmad and R. Harrison, "Industrial cyber physical systems: A survey for control-engineering tools," *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, St. Petersburg, 2018, pp. 270-276. doi: 10.1109/ICPHYS.2018.8387671

[26] C. S. Shankar, A. Ranganathan and R. Campbell, "An ECA-P policy-based framework for managing ubiquitous computing environments," *The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, San Diego, CA, USA, 2005, pp. 33-42. doi: 10.1109/MOBIQUITOUS.2005.11

[27] V. Gunes, S. Peter, T. Givargis and F. Vahid, "A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems," *KSII Transactions on Internet and Information Systems*, vol. 8, no. 12, pp. 4242-4268, 2014. DOI: 10.3837/tiis.2014.12.001

[28] G. Gardašević, M. Veletić, N. Maletić, D. Vasiljević, I. Radusinović, S. Tomović, and M. Radonjić, “The IoT Architectural Framework, Design Issues and Application Domains,” *Wireless Personal Communications*, vol. 92, no. 1, pp. 127–148, Jan. 2017.

[29] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated. 2007.

[30] X. Cao, P. Cheng, J. Chen and Y. Sun, "An Online Optimization Approach for Control and Communication Codesign in Networked Cyber-Physical Systems," in *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 439-450, Feb. 2013. doi: 10.1109/TII.2012.2216537

[31] M. Barcelo, A. Correa, J. Llorca, A. M. Tulino, J. L. Vicario and A. Morell, "IoT-Cloud Service Optimization in Next Generation Smart Environments," in *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 4077-4090, Dec. 2016. doi: 10.1109/JSAC.2016.2621398

[32] T. Yu, B. Zhou, K. W. Chan, L. Chen and B. Yang, "Stochastic Optimal Relaxed Automatic Generation Control in Non-Markov Environment Based on Multi-Step$Q(\lambda)$Learning," in *IEEE Transactions on Power Systems*, vol. 26, no. 3, pp. 1272-1282, Aug. 2011. doi: 10.1109/TPWRS.2010.2102372

[33] P. E. Guerrero, K. Sachs, S. Butterweck, and A. Buchmann, “Performance Evaluation of Embedded ECA Rule Engines: A Case Study,” *Computer Performance Engineering Lecture Notes in Computer Science*, Springer Publishing, pp. 48–63, 2008.

[34] R. Milo, “Network Motifs: Simple Building Blocks of Complex Networks,” *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[35] R. Storn and K. Price, "Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces", *Journal of Global Optimization*, 1997. https://doi.org/10.1023/A:1008202821328

[36] J. Brownlee, *Clever algorithms: nature-inspired programming recipes*. 2012. [Online]. Available: http://www.cleveralgorithms.com/

[37] D. Davendra, I. Zelinka, R. Senkerik, M. Pluhacek, “Complex Network Analysis of Evolutionary Algorithms Applied to Combinatorial Optimisation Problem,” in

Proceedings of the Fifth International Conference on Innovations in Bio-Inspired Computing and Applications IBICA 2014, Cham, 2014.

[38] R. N. Khushaba, A. Al-Ani, and A. Al-Jumaily, " Feature subset selection using differential evolution and a statistical repair mechanism," *Expert Syst. Appl.* 38, 9, 11515-11526. DOI=http://dx.doi.org/10.1016/j.eswa.2011.03.028, September, 2011.

[39] C. Lin, M. Han, Y. Lin, S. Liao and J. Chang, "Neuro-fuzzy system design using differential evolution with local information," *2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011)*, Taipei, 2011, pp. 1003-1006. doi: 10.1109/FUZZY.2011.6007522

[40] S. Guo, C. Yang, P. Hsu and J. S. Tsai, "Improving Differential Evolution With a Successful-Parent-Selecting Framework," in *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 5, pp. 717-730, Oct. 2015. doi: 10.1109/TEVC.2014.2375933

[41] J. S. Arora, *Introduction to optimum design*. Amsterdam: Elsevier/Academic Press, 2017.

[42] N. Cheney, R. MacCurdy, J. Clune, and H. Lipson, "Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding," *SIGEVOlution* 7, 1 11-23. August, 2014. DOI=http://dx.doi.org/10.1145/2661735.2661737

[43] S. Takahashi, H. Kita, H. Suzuki, T. Sudo and S. Markon, "Simulation-based optimization of a controller for multi-car elevators using a genetic algorithm for noisy fitness function," in *The 2003 Congress on Evolutionary Computation*, 2003. CEC '03., Canberra, ACT, Australia, 2003, pp. 1582-1587 Vol.3. doi: 10.1109/CEC.2003.1299861

[44] H. Takagi, "Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation," in *Proceedings of the IEEE*, vol. 89, no. 9, pp. 1275-1296, Sept. 2001. doi: 10.1109/5.949485