2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys) | 978-1-7281-7651-2/20/$31.00 ©2020 IEEE | DOI: 10.1109/DEPENDSYS51298.2020.00017

Real-Time Fault Detection and Diagnosis of CPS Faults in DEVS

Joseph Boi-Ukeme
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON, Canada
joseph.boiukeme@carleton.ca

Cristina Ruiz-Martin
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON, Canada
cristinaruizmartin@sce.carleton.ca

Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON, Canada
gwainer@sce.carleton.ca

***Abstract*— The technological advancement in Cyber-Physical Systems (CPS) has evolved into sophisticated hardware, leading to systems that are complex and interconnected. This trend has made modern CPS susceptible to faults. Traditional methods for fault detection and diagnosis are unable to adequately scale up to manage the faults that occur in CPS because of the tight interconnectivity between the physical and cyber parts of CPS. Also, hard real-time constraints present challenges that are not sufficiently addressed by traditional fault-tolerant design approaches; therefore, new methods are needed to deal with these faults. Here, we present a new scheme to detect and diagnose CPS faults, which relies on the combination of knowledge-based and model-driven Fault Detection and Diagnosis (FDD). The method is developed to be applied when building CPS using Discrete Event Methodologies.**

Keywords—FDD, Cyber-physical systems, DEVS

I. INTRODUCTION

The steady advancement in sensing, actuation, and control system computing technologies has led to the development of complex systems, including smartphones, autonomous vehicles, smart grids, automated buildings, smart cities [1]. The common characteristic of these systems is the tight coupling of hardware and software capabilities and constraints as well as the combination of heterogeneous components. These devices have hardware with advanced sensing and actuation with embedded computing capabilities. Similarly, the software now includes tools for data management, learning methods, and optimization [2]

This level of complexity has necessitated the development of theoretical and practical methods for developing CPS. The main objective of the theory of CPS is to ensure that the interaction between software and hardware designs is better coordinated. Better coordination is achieved through improved awareness of the constantly evolving environment and improved capacity to handle huge amounts of data. [3]

Current CPS are such that their designs are combinations of several subsystems and can become large and difficult to manage. Similarly, there is a tight interconnection between the physical and cyber parts, and these are closely designed together, which adds another level of complexity. In addition to the complexity of the design process, modern-day CPS are fully or partially autonomous with multiple and diverse applications. The literature in the field shows a wide acceptance of the benefits that automated control brings to manage this complexity. However, with increased complexity, tight interconnection, and automation, CPS are more susceptible to faults, making them more fragile [3][4].

Despite the complexity inherent in the design of CPS, the sensitivity of the applications of CPS require that they must be fault and failure tolerant. To design failure and fault-tolerant CPS, design methods must be able to anticipate the occurrence of faults, detect faults, and respond to these faults thereby preventing them from escalating to failure. Fault detection and diagnosis methods, when incorporated into the design of CPS, are computationally intensive. Therefore, they should be event-triggered and must be able to differentiate between faults and uncertainties.

With this panorama in mind, we will propose a generic fault detection and diagnosis scheme capable of diagnosing CPS faults in real-time. The scheme is developed to accommodate different modeling methods but for clarity of explanation, we adapt it to the Discrete Event System Specification (DEVS) formalism. To assess the scheme, we implemented a library to store fault codes in a data structure and developed intelligent logic to ensure faults are correctly detected and isolated.

II. BACKGROUND

The main goal of Fault Detection and Diagnosis (FDD) is to correctly determine the nature, extent, location, and time of detection of a fault, based on available measurements obtained from the system [5]. To

978-1-7281-7651-2/20/$31.00 ©2020 IEEE
DOI 10.1109/DependSys51298.2020.00017

properly detect and diagnose a fault, several researchers have proposed the following sequence: firstly, execute a fault detection step, where faults are detected using any mechanism prescribed by the system designers; after a fault has been detected, isolate the fault and determine the physical location of the fault; then, determine the nature of the fault and determine the extent of the impact of that fault in a process called fault identification and risk assessment [3][6].

There have been many methodologies proposed for FDD. However, FDD in CPS is challenging. The complex nature of CPS makes it difficult to distinguish faults from uncertainty (noise) and we need robust FDD to avoid false alarms. Faults in CPS can affect multiple components, therefore, FDD in CPS must be able to handle multiple faults. In large CPS, it is difficult to isolate faults that occur within their sub-systems and when these faults occur, they could quickly escalate to failures; hence, another challenge is to ensure that the faults are detected and isolated promptly [3]

Timeliness is particularly important for safety-critical CPS systems in which the control of the CPS is done with the aid of sensors, a control algorithm, and actuators. In these systems, failing to meet timing deadlines can have significant effects. Undetected faults can affect timeliness, leading to potential system instability with catastrophic consequences [7]

There have been different methods proposed for FDD for CPS, which can be classified based on three major paradigms, viz., model-based, data-driven/knowledge-based, and hardware-based [8]. Several methods have been proposed for each of these paradigms and it is important to note that they are not mutually exclusive but, in the following, we will adopt this classification.

- **Model-Based FDD:** they are based on the use of a model developed based on some of the properties of CPS physics. These models can be qualitative, in which input-output relationships are expressed as qualitative functions; or quantitative, where the input-output relationship is expressed in terms of a mathematical function. [9].

- **Data-Driven/Knowledge-Based FDD:** they use models obtained from known input and output CPS data. This generates a data-driven model of the process. The model is then compared with real-time process data to find faults.

- **Hardware-Based FDD:** we could use dedicated hardware or hardware integrated as part of the CPS. Typically, methods that fall within this category do not use models (unlike the other two methods).

Model-Based FDD methods are the most effective in detecting and diagnosing unknown faults because the method does not depend on large real-time data for its detection and diagnosis. On the downside, the models used must define the input/output relationships accurately, and not all modeling techniques are adequate for this. The Knowledge Based FDD methods are particularly good for real-time FDD, however, they require large sets of historical data which may increase computational complexity. In this work, we explore the advantages of combining the knowledge-based and model-based FDD approaches [10].

As discussed in the introduction, we are interested in applying these methods to the development of CPS using the DEVS formalism, a formal modeling methodology based on systems theory [11]. The DEVS formalism decomposes complex systems into atomic and coupled models, where the atomic models specify the behavior, and the coupled model specifies the structure [12]. In DEVS, a CPS can be modeled as a composite of atomic and coupled models. DEVS provides a rich structural representation of components and can explicitly specify timing, which makes it easily adaptable for real-time systems. Various real-time systems have been successfully developed using DEVS [13][14][15].

The application of DEVS for real-time systems adds another level of difficulty because real-time constraints require that models now must interact with the environment in real-time. The environment could include software, hardware components, or human operators. DEVS by default is unable to manage this for a few reasons. DEVS uses virtual time (periods of inactivity are skipped) and we need to use a real-time clock for real-time CPS. The computing platforms used can affect the physical time it takes to execute a model, which creates a disparity between a simulated model and what happens in the physical system. Similarly, state transitions in DEVS events are defined to occur instantaneously (e.g., an event is defined as taking zero time); however, in real-time simulation state changes and operations may occur during a time interval (or a time window). Finally, it is difficult to validate simulated models in the real world [16][17]

To address the problem of adapting DEVS for real-time systems, different researchers have produced different approaches that fall under two categories: new design methods for modeling real-time systems (for example, extending the DEVS formalism) or executing logical models using real-time DEVS simulators.

Different approaches extend the DEVS formalism; for instance, [18] introduced the use of time windows or the concept of uncertainty intervals explored in [19]. A time window is a function that can take a value within a range instead of a single instant like the time advance; this allows events to occur within a period. In [17], Real-Time DEVS (RT-DEVS) was proposed; RT-DEVS is a major

milestone because other approaches that extended the DEVS formalism have some similarities with RT-DEVS. In RT-DEVS, real-time execution of models is facilitated by the addition of time interval functions (time-windows). With time windows, we can restrict the simulation time to ensure real-time execution. The concept of activity specification with constraints is defined for each state, which means some computation can be done without any modification to the state. The drawback of the RT-DEVS formalism is that activities are not explicitly included in the internal and external transition functions making it impossible to prioritize activities and it does not support receiving multiple inputs and sending multiple outputs because it is based on Classical DEVS. To tackle these drawbacks, [16] proposed the Action Level RT DEVS, which specifies constrained and schedulable actions in addition to individual state changes. Other researchers that employed an RT-DEVS simulator include DEVS on a chip [20], ECD++ [21], Power DEVS [22], and ECDboost [23]. The advantage of adding real-time functionalities to a DEVS simulator is that it does not add an extra level of complexity for the modeler and models developed using these simulators are backward compatible with existing DEVS models.

In analyzing the definition of a fault, it is important to note that when the model has been designed and the modeler considers it is correct, it will not normally change. However, when this model is deployed to the target platform as a control software using DEVS which we call the Discrete Event Control Software (DECS), the DECS could violate the specification originally described by the model. From the foregoing, we will say that a fault in a DEVS-based CPS has occurred when the behavior of the CPS does not conform to the specification defined in the DEVS model.

III. GENERIC FDD IN DEVS

In this section, we propose a generic FDD scheme to detect and isolate faults. The scheme is designed to satisfy the following requirements:

1. Correctly identify faults and distinguish them from noise or uncertainty
2. Isolate faults within a complex CPS
3. Manage multiple faults
4. Prevent faults within subsystems from escalating into major faults
5. Consume minimum resources and have fast computation.

The scheme is event-based, and it only triggers the fault tolerance algorithm when the values that fall out of range within the CPS are confirmed to be faults. This is because not all values that fall out of range are faults. It could just noise or uncertainty. Within the CPS model, we would define algorithms for fault tolerance. These algorithms would only be executed when a fault has been confirmed by the FDD scheme presented in this section. The scheme is shown in Fig. 1.

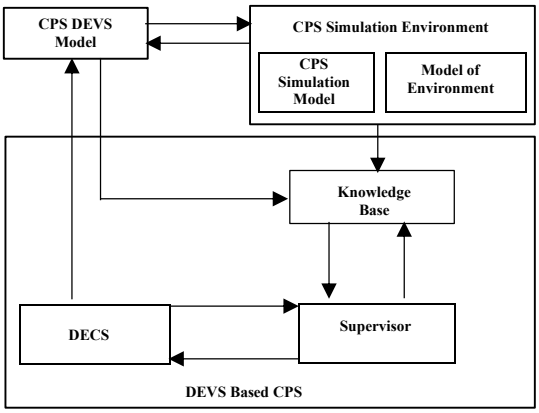

Fig. 1. Proposed CPS FDD Scheme

The first component is the ***CPS DEVS Model***. It is a model of the CPS using DEVS, which is used for testing and to populate the Knowledge Base with faults. Once this model has been specified and implemented, we can execute it in a simulation environment: the ***CPS Simulation Environment*** component. This is a model of the environment of the CPS that allows us to simulate the CPS DEVS Model. With these simulations, we can study the model's behavior in various scenarios. Based on the simulation results, we can go back to the CPS DEVS Model, modify the specifications if needed, and run new simulations. We also use the CPS Simulation Environment to study the model for possible faults. These studies would provide data to populate the ***Knowledge Base*** component. Once we are satisfied with the simulation results, the CPS DEVS Model, which was already evaluated using the CPS Simulation Environment, is deployed into the target platform and it is transformed into the *DECS.*

The ***DECS*** component is deployed into the target platform for operational purposes. After the model has been ported to the target platform, we perform further testing and calibration. We may need to go back and redefine the CPS DEVS Model as the testing of the DECS in the real world or the CPS Simulation Environment may reveal some potential faults or design errors that were not properly captured in the initial specification. In that case, we would always go back to the CPS DEVS Model and modify it. Every modification should be done in the original model. Any fault information observed during testing and calibration of the DECS is stored in the Knowledge Base component.

The ***Knowledge Base (KB)*** is a database that holds information about the known set of faults that can occur in the CPS. The Knowledge Base evolves through the

lifetime of the CPS because as new faults become known, information about these faults would be updated in the Knowledge Base. The information about faults is obtained from the simulation of the CPS DEVS Model in the CPS Simulation Environment, and in the experiments performed on the DECS. The supervisor communicates with the Knowledge Base to confirm faults.

During its operation, if a fault occurred, the DEVS Based CPS would send the fault information to the ***Supervisor***, an intermediate component between the DEVS Based CPS and the Knowledge Base that manages the fault detection process. When the DEVS Based CPS notifies values out of range, it confirms with the Knowledge Base if a fault has occurred. If a fault occurred, it notifies the DEVS Based CPS to take the recommended actions about that fault.

In the rest of the section, each component that makes up the FDD scheme shown in Figure 1 is explained in more detail.

A. CPS DEVS Model

The first step for developing the CPS DEVS Model is understanding the CPS of interest and defining a requirements document about the expected behavior of the system. Deviations from the expected behavior are going to be considered faults, and they will be added to the Knowledge Base. With the information obtained from the requirements, we build a model of the CPS, including mechanisms to report deviations in the expected behavior. The model would be designed to send a special type output whenever these values fall out of range, called fault messages (*fm*). The *fm* is a unique code with information about the atomic model and the state variable out of range. The structure of the *fm* message is like the *fault_id* in the Knowledge Base, which would be described in section D. We do not call it *fault_id* because it is not yet confirmed as a fault from the Knowledge Base. While an *fm* can be a fault or uncertainty, a *fault_id* will be used for actual faults.

After the model has been specified, we would test it using the CPS Simulation Environment. After we are satisfied with those tests, we would port it to the target platform where more tests and experiments are done. The tests and experiments conducted in both the simulation environment and the target platform can yield results that will result in a redesign of the model.

B. CPS Simulation Environment

The CPS Simulation Environment allows us to evaluate the CPS DEVS Model. During this testing phase, we confirm that the model behaves as expected. We also deliberately inject possible faults (for example feeding the model with inputs that are out of range to observe how the model responds to these faults. This provides more information about faults. It also allows us to improve the model with different methods to respond to these faults. Once we are convinced with the simulation results, we port the CPS DEVS Model to the target platform. The model deployed into the target platform is the DECS detailed in section C. Information about faults discovered during the simulations is used to populate the Knowledge Base component described in section D.

C. DECS

The CPS DEVS Model ported to the target platform and now operating in its environment is the DECS. This is controlling a physical system in its operating environment and thus it is prone to faults and uncertainties. Therefore, after the models have been ported to the target platform, we conduct more tests. If new faults are discovered during them, we may redefine the CPS DEVS Model, and we would include the information about these new faults in the Knowledge Base. After testing is complete, the DECS is now ready for the operating environment.

During its operational phase, the DECS communicates with the Supervisor through its physical output ports. The DECS would send an output fault message (*fm*) to its physical output port whenever there is a value or a set of values out of range. As the DECS is the CPS DEVS Model in the target platform, *fm* is the message earlier explained. Each component specified as a DEVS atomic model would be able to generate an *fm* to indicate that there is a value out of range, and therefore, there may be a fault within the CPS.

D. Knowledge Base

The Knowledge Base is a database that holds information about faults that can occur in the DEVS Based CPS. To initially populate the Knowledge Base, this information is obtained from the following sources:

- **CPS DEVS Model:** possible faults that can occur identified during the analysis of the system and the design of the model
- **CPS Simulation Environment:** faults identified during experimentation on the simulated environment.
- **DEVS Based CPS:** Experimentation and testing operation of the CPS within its operating environment

The Knowledge Base is updated whenever new faults are identified using the actual data retrieved from the DEVS Based CPS operation. It is important to remark that the Knowledge Base would evolve through the life cycle of the CPS. Not all faults can be captured in the design stage, therefore if new faults appear during the operation of the CPS, they will be added to the Knowledge Base.

The faults in the Knowledge Base are associated with a frequency of occurrence. The frequency of occurrence is updated using the faults that occur during the CPS daily operations.

The information stored in this component is important for two reasons. Firstly, it would be a source of information to update the models if needed; for example, if we obtain fault information from a test or the physics of the CPS that was not captured in the original design, we can use this information to update our CPS DEVS Model. Secondly, it is a critical component in the FDD schema because we will use this information to confirm the presence or absence of a fault in the CPS.

To ensure proper operation of the FDD scheme, the Knowledge Base component must meet the following requirements:

• **Minimize memory usage**: Because CPS have memory constraints, the added FDD should minimize memory usage to ensure it does not degrade the performance of the original CPS.

• **Allow for quick and reliable search:** The real-time requirement of the CPS makes any FDD scheme useful only when it can produce a quick and accurate result.

• **Mutable:** The Knowledge Base can be updated at any time during the life cycle of a CPS. This is important to allow us to include new faults that were not discovered during design and experimentation.

• **Unique Fault ID:** Each fault store in the Knowledge Base must have a unique and consistent fault id. To achieve these requirements, we should follow a specific convention - for example, A12. This will allow us to quickly isolate faults by simply looking at this *fault_id*

One of the requirements is a consistent pattern for the *fault_id.* To obtain a consistent fault id, we can use, for example, an alphanumeric coding convention (e.g. A1). Using this convention, we will label the atomic models in the DECS as alphabets and the state variables in terms of a number, hence A1 would mean, atomic model A and state variable 1 within the atomic model. Typical faults within a CPS model are not localized in one atomic model. A fault code can indicate that state variables from more than one atomic model are out of range. For example, A1B1 would mean that both atomic models A and B have the state variable 1 out of range. As we detail in the next section, the Supervisor oversees generating these combined codes. The alphanumeric convention is just an example. Other conventions may be used if they provide unique fault ids and allow us to quickly isolate faults through the fault id.

To satisfy the other requirements, different implementations could be adopted depending on the size of the CPS. To keep modularity, a clear separation of components, and boost reusability, the Knowledge Base component is defined as a DEVS model that manages a database. The database will be implemented in different formats based on the application. For example, we can define the database as a plain text file where each row represents a fault and the associated frequency of occurrence. We can also define it as a set of two data structures where the first one would be a bloom filter to store the faults in a manner in which we can quickly confirm if a fault is present or not, and the second one is a standard hash to store the details of the faults and frequency of occurrence that can be retrieved if the presence of that fault has been confirmed. The bloom filters are data structures that do not store actual values hence they are relatively fast and require low storage and the storage is usually fixed and does not change when more data is added to the filter. They are also good because they do not return false negatives and have a low probability of false positives. An additional useful property of the bloom filter is that when data is stored, no deletion is allowed.

The Knowledge Base component is formally defined using DEVS as the following atomic model:

```
KB = <S, X, Y, δint, δext, δcon, λ, ta>
S = {s1 ∈ {notify, passive} kb_match ∈ {True,
False} DB = associated database}
X = {fault_id ∈ string}
Y = {True, False}
δint (s) = {if s1 = notify then s1 = passive}
δext (s,e,x) = {
    if s1 = passive, then
        Check if X (i.e. fault_id) exists in DB
            If fault_id exist, then
                kb_match = True
                increment Freq of fault_id DB
            else
                kb_match =False
            end if
        s1 = notify
    else
        //This case should never be reached   }
δcon = δint + δext
λ (s1=notify, kb_match, DB) = {kb_match}
ta (s1 = passive, kb_match, DB) = INFINITY
ta (s1 = notify, kb_match, DB) =
checktime}
```

Regardless of the implementation of the database, when the Knowledge Base receives an input with a *fault_id*, if the *fault_id* is in DB, it will update the frequency of the *fault_id* in the database (DB) and update its state variable *kb_match* to True. Otherwise, *kb_match* will be updated to false. In any case, the state variable s1 will be updated to *notify*. After the *checktime* (i.e., the time it takes the Knowledge Base to confirm the presence of a fault in its database), it will output the value of the *kb_match* variable and update its state variable s1 to

passive. The model will remain passivated until a new *fault_id* is received.

E. Supervisor

The purpose of the Supervisor is to ensure that the process of FDD between the DECS and the Knowledge Base is event-driven. The Supervisor would watch all the fault outputs from various atomic models to determine which part of the entire DECS sends an *fm* signal indicating a likely fault. The *fm* from the atomic models does not necessarily mean a fault has occurred. A fault has occurred only when the Supervisor has confirmed the *fault_id* from the Knowledge Base, a fault message *(fm),* or a combination of fault messages *(fms)* from the *fault_id*. By looking at the *fm*, the supervisor can determine where the fault signal is coming from, which is useful in isolating faults. On receiving an *fm* or multiple *fm* messages, the supervisor creates the *fault_id* and sends a request to the Knowledge Base to confirm that this is a fault. Once the supervisor confirms that a CPS fault has occurred, we can initiate any prescribed action for that fault within the atomic model responsible for that part of the CPS. In summary, the supervisor receives a fault message, creates a *fault_id* following a predefined convention, requests a check from the Knowledge Base and if the fault is present, it sends an output to the affected atomic model.

The Supervisor is defined as a DEVS model with two input ports and two output ports. In the first input port, it receives fault messages (*fm*), while in the second input port it receives the confirmation of faults (i.e. true/false). Its first output port is used to send a check request (i.e. a *fault_id*) while the second output port is used to send an output trigger to confirm the presence of a fault.

The Supervisor component is formally defined using DEVS as the following atomic model:

```
SUPERVISOR = <S, X, Y, δint, δext, λ, ta>
S = {passive, check, key, out_key, accommodate}
X = {fm =fault_id, match = {True, False}}
Y = {chk ∈ string, read∈ string}
δint (S) = {
    if S = check then S = passive
    else if S = accommodate then S = passive
    else if S = check then S = passive   }
δext (fm, state == passive) {
    If fm does not exist
        key = fm
    else
        key = composed (fm)
    S = check    }
δext (match is True, S == passive)
    { out_key = matched_id
    S = accommodate   }
δext (match = false, S == passive) {
    // do_nothing}
δext (match = true, S == check) {S = check}
δext (fm, S == accommodate) {S = accommodate}
λ(check) {send key to read}
λ(accommodate) {send out_key to chk}
ta(passive) = INFINITY
ta(accommodate) = triggerttime
ta(check) = checkingtime
```

IV. Case Study of FDD

To show the usability of the FDD scheme presented in this paper, we design a simple example where we explain the major components of the FDD scheme. In this example, for simplicity, we have designed, tested, run simulations, and experiments on a CPS DEVS model. After completing the design, we discovered that the CPS has yielded seven types of faults and we have uniquely defined the *fault_ids* and stored their values in the Knowledge Base presented in Table I.

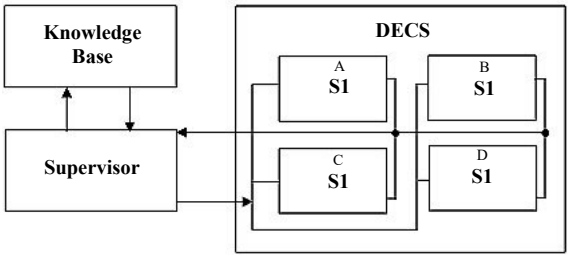

Fig. 2. Model Structure for FDD scheme

We also obtained information about the nature of the faults that would help us set the priorities of the various fault messages from the atomic models that make up the CPS. The DECS, Knowledge Base, and Supervisor are shown in Fig. 2. The figure shows three (3) blocks that are from the FDD scheme presented in section III. In this figure, the DECS has been expanded to show its component models which are discussed in the subsequent sections.

A. DECS

In this case, we use four (4) fault generators (Model A, Model B, Model C, and Model D) which are modeled using DEVS to produce faults in a random pattern which is consistent with the way a CPS fault would present itself. These models would each have one state variable each A1, B1, C1, and D1, respectively. We define a possible fault as a state variable that is out of range and produces a message.

Every time the model generates a message, it sends an output to the supervisor. It is important to note at this point that the fact that a state variable is out of range does not mean that a fault is present, every fault would need to be validated from the Knowledge Base.

B. Knowledge Base

For small CPS with minimum memory requirements, we implemented the structure of the Knowledge Base as a flat-file; the supervisor can read the content of the file into memory and write to the file after a specified period. This type of Knowledge Base would work well for simple CPS; however, when the systems get large and complex there would be performance issues as the size of the file can grow large which would lead to slow and resource-intensive read and write. To deal with this, we implemented the Knowledge Base as a DEVS model that is initialized with the known faults and communicates through its inputs and outputs to the rest of the system. The Knowledge Base receives an input in the form of a *fault_id* and stores the fault information in a bloom filter data structure, this would help us maintain consistent performance, however, since the bloom filter does not store the actual fault codes, it is difficult to retrieve the frequency of occurrence of faults, and therefore an additional data structure would be needed.

Table I. describes the initial state of the Knowledge Base we use in the example presented in this paper. The first two columns are part of the Knowledge Base structure and the third column is just a description of what the fault code means for readability purposes. The seven different fault codes have zero occurrences at the initial design stage before the start of the simulation.

TABLE I. THE INITIAL STATE OF THE KNOWLEDGE BASE

Fault id	Frequency	Description
A1	0	Fault indicated A1 being out of range
A1B1	0	Fault indicated by A1 & B1 being out of range
A1C1	0	Fault indicated by A1 & C1 being out of range
A1D1	0	Fault indicated by A1 & D1 being out of range
B1C1	0	Fault indicated by B1 & C1 being out of range
C1	0	Fault indicated by C1 being out of range
C1D1	0	Fault indicated by C1 & D1 being out of range

C. Supervisor

The Supervisor would watch the fault outputs from the four atomic models (A, B, C, and D) to determine which part of the DECS an *fm* signal indicating a likely fault originates from. On receiving an *fm* or multiple *fm* messages, the supervisor creates the *fault_id* based on the specification defined in section III. Then it sends a request to the Knowledge Base to confirm that this is a fault. Once the supervisor receives a confirmation that a CPS fault has occurred from the Knowledge Base, we send a signal to the atomic model affected. In this implementation, we are not concerned about how the model accommodates the fault. We are only interested in detecting and isolating the faults. Prescribed action for fault accommodation is reserved for future work.

D. Results

This section describes the results obtained from the implemented Knowledge Base using the data presented in Table I, the DEVS based CPS and Supervisor explained earlier. Fig. 3 shows the logs from four models, which represent the fault messages for one simulation run for ten seconds. There are 4 plots for model A, B, C, and D respectively, the horizontal axis shows simulation time in seconds, the vertical axis shows the outputs of the atomic models with each spike representing a fault message.

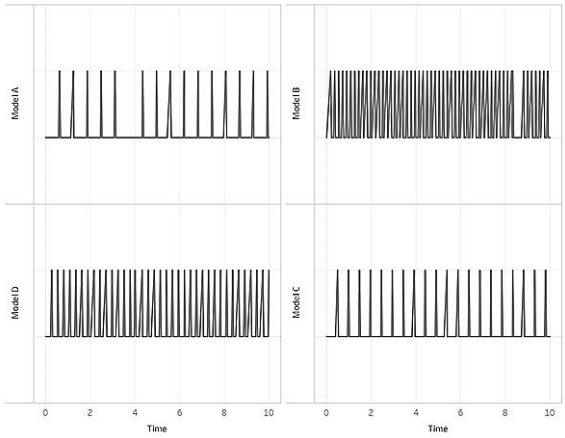

Fig. 3. Simulation Logs from Fault Messages

From the logs, we can observe the number of times each model had a value that exceeded the specified range. For this simulation run, model A has a count of 15, model B 54, model C 20, and model D 37. The supervisor checked the Knowledge Base and found the following faults which are tabulated after simulation, the final output of our model is the frequency of the faults and this is equivalent to the number of times the supervisor sends a trigger to each atomic model in a faulty state. This is shown in Table II.

TABLE II. THE FINAL STATE OF THE KNOWLEDGE BASE

Fault Code	Frequency
A1	15
A1B1	4
A1C1	2
A1D1	2
B1C1	5
C1	20
C1D1	2

From the results presented in Fig. 3. and Table II, the supervisor received a total of 126 fault messages from the atomic models. The total number of actual faults confirmed by the supervisor was 50. This shows that the

scheme was able to distinguish between a fault and uncertainty because 76 of those messages were an indication of transient disturbance. Similarly, from the scheme, each atomic model fault was sent within the same time window and some messages appear at the same time representing a scenario of multiple faults. The results show that in such a situation, with the priority method in the Supervisor, the Supervisor can react to these multiple faults. From Table II, by simply looking at the *fault_id*, we can tell which atomic model(s) and state variable the fault originated from, this convention serves two purposes. Firstly, it helps us quickly determine the location of the fault, and secondly, the fact that it is detected in the atomic model is early enough to prevent the faults from escalating.

V. Conclusion and Future Work

In this work, we presented a fault detection and diagnosis (FDD) scheme that is capable of detecting and isolating faults in real-time, keep track of the frequency faults, handle multiple faults when they occur, and prevent false alarms by validating probable faults from the Knowledge Base. The scheme is based on a knowledge-based and model-based approach. The scheme presented is consistent with the DEVS formalism and is backward compatible with existing CPS models built using DEVS. We also discussed various options for scaling up the FDD scheme to more complex CPS. The FDD scheme presented here is simple to implement, however, CPS are complex and could generate more faults that may reduce the performance of the FDD scheme. Our future work includes deploying the FDD scheme to a target platform to test various options for physical storage of the Knowledge Base and apply the options for scaling the scheme to more complex applications to test its performance. One way to achieve this is to have the FDD scheme in modules in different parts of the CPS, this modular approach can be done in two ways, one would be to have the complete FDD scheme in different parts of the system or a second approach would be to have multiple small Knowledge Bases but one supervisor.

References

[1] S. Chaterji, P. Naghizadeh, M. A. Alam, S. Bagchi, M. Chiang, D. Corman, B. Henz, S. Jana, N. Li, S. Mou et al., "Resilient cyberphysical systems and their application drivers: A technology roadmap," arXiv:2001.00090, 2019.

[2] S. Weerakkody, O. Ozel, Y. Mo, and B. Sinopoli, "Resilient Control in Cyber-PhysicalSystems: Countering Uncertainty, Constraints, and Adversarial Behavior," *Foundations and Trends® in Systems and Control*, vol. 7, no. 1-2, pp. 1–252, 2020.

[3] V. Reppa, M. M. Polycarpou, and C. G. Panayiotou, "Sensor Fault Diagnosis," *Foundations and Trends® in Systems and Control*, vol. 3, no. 1-2, pp. 1–248, 2016.

[4] Kröger Wolfgang and E. Zio, *Vulnerable Systems*. London: Springer London, 2011.

[5] S. Katipamula and M. Brambley, "Review Article: Methods for Fault Detection, Diagnostics, and Prognostics for Building Systems—A Review, Part I," *HVAC&R Research*, vol. 11, no. 1, pp. 3–25, 2005.

[6] M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki, *Diagnosis and fault-tolerant control*. Berlin: Springer, 2016.

[7] L. Sherry and R. Mauro, "Controlled Flight into Stall (CFIS): Functional complexity failures and automation surprises," *2014 Integrated Communications, Navigation and Surveillance Conference (ICNS) Conference Proceedings*, Herndon, VA, 2014, pp. D1-1-D1-11.

[8] A. Mouzakitis, "Classification of Fault Diagnosis Methods for Control Systems," *Measurement and Control*, vol. 46, no. 10, pp. 303–308, 2013.

[9] Z. Gao, C. Cecati and S. X. Ding, "A Survey of Fault Diagnosis and Fault-Tolerant Techniques—Part I: Fault Diagnosis With Model-Based and Signal-Based Approaches," in *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3757-3767, June 2015.

[10] S. Lazarova-Molnar, H. R. Shaker, N. Mohamed, and B. N. Jorgensen, "Fault detection and diagnosis for smart buildings: State of the art, trends, and challenges," *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, Muscat, 2016, pp. 1-7.

[11] B. P. Zeigler, Prähofer Herbert, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. San Diego: Academic Press, 2000.

[12] G. A. Wainer, *Discrete-event modeling, and simulation: a practitioner's approach*. CRC Press, 2009.

[13] H. S. Song and T. G. Kim, "Application of Real-Time DEVS to Analysis of Safety-Critical Embedded Control Systems: Railroad Crossing Control Example," *Simulation*, vol. 81, no. 2, pp. 119–136, 2005.

[14] B. Earle, K. Bjornson, J. Boi-Ukeme and G. Wainer, "Design and Implementation of a Building Control System in Real-Time Devs," *2019 Spring Simulation Conference (SpringSim)*, Tucson, AZ, USA, 2019, pp. 1-12

[15] C. Ruiz-Martin, A. Al-Habashna, G. Wainer and L. Belloli, "Control of a Quadcopter Application with Devs," *2019 Spring Simulation Conference (SpringSim)*, Tucson, AZ, USA, 2019, pp. 1-12.

[16] H. S. Sarjoughian and S. Gholami, "Action-level real-time DEVS modeling and simulation," *Simulation*, vol. 91, no. 10, pp. 869–887, 2015.

[17] J. S. Hong and T. G. Kim, "Real-time discrete event system specification formalism for seamless real-time software development", *Discrete Event Dynamic Systems: Theory Appl.*, vol. 7, no. 4, pp. 355-375, 1997.

[18] Q. Wang and F. E. Cellier, "Time Windows: Automated Abstraction of Continuous-Time Models into Discrete-Event Models in High Autonomy Systems∗," *International Journal of General Systems*, vol. 19, no. 3, pp. 241–262, 1991.

[19] D. Vicino, O. Dalle, and G. Wainer. 2015. "Using finite forkable DEVS for decision-making based on time measured with uncertainty," *8th International Conference on Simulation Tools and Techniques,* Athens, Greece, 2015 pp. 89–98.

[20] Xiaolin Hu, B. P. Zeigler and J. Couretas, "DEVS-on-a-chip: implementing DEVS in real-time Java on a tiny Internet interface for scalable factory automation," *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*, Tucson, AZ, USA, 2001, pp. 3051-3056 vol.5.

[21] Y.H. Yu and G. A. Wainer, "ECD++: an engine for executing DEVS models in embedded platforms," *2007 Summer Computer Simulation Conference (SCSC '07)*, San Diego, CA, USA, 2007, pp.323–330.

[22] F. Bergero and E. Kofman, "PowerDEVS: a tool for hybrid system modeling and real-time simulation," *Simulation*, vol. 87, no. 1-2, pp. 113–132, 2010.

[23] G. Wainer and R. Castro. "DEMES: A Discrete-Event methodology for Modeling and simulation of Embedded Systems," *Modeling and Simulation Magazine*, 2, pp.65-73, 2011.