International Journal of Computer Applications (0975 - 8887)
Volume 176 - No.5, October 2017

Automatic Test Scenario Generation Technique for Medical Cyber Physical Systems

Afrina Khatun
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

Naushin Nower
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

ABSTRACT

Medical Cyber Physical Systems (MCPS) are life-critical, context-aware, and networked systems of software controlled medical devices, that are responsible for monitoring and controlling the physical dynamics of patient's bodies. MCPS are designed to provide high-quality continuous care for patients in complex clinical scenarios, and also thus arise various safety issues compare with the traditional medical systems. Controller software makes important decisions in MCPS, that can directly affect peoples lives and thus it is needed to validate and verify the whole MCPS system to ensure patient's safety. Although various techniques have been proposed in literature to ensure safety of MCPS, most of them follow model based simulation and do not provide any test cases for logically testing the whole systems. An automated test scenario generation approach for assisting the task of safety assurance of MCPS has been proposed in this paper. The proposed technique takes system state models as a input and extracts necessary information to generate state graph. In the next step, it extracts all possible system state transitions. And, finally generates test scenarios based on the extracted paths and stores them in text documents for the tester. The proposed technique has been applied on a Generic Patient Controlled Analgesia Pump Model and has been successful to generate test scenarios covering all the transitions of the state models. Thus, it automatically generates all the test cases and by testing all the paths in the system can ensure the safety of patients.

Keywords

Medical Cyber Physical Systems, UML diagram, Test case generation, Patient's safety, GPCA

1. INTRODUCTION

Cyber Physical Systems (CPS) are a collection of computational (cyber) and physical components that interact with each other to achieve a particular objective within a specific time frame. CPS enable the virtual world to interact with the physical world in order to monitor and control the intended parameter [1]. Because, of this interaction the applicability of CPS is found in numerous time-critical applications from smart house to smart grid. Emerging applications of CPS include medical devices and systems, aerospace systems, transportation vehicles and intelligent highways, defense systems, robotic systems, process control, factory automation, building and environmental control, smart spaces, intelligent home and so on [2].

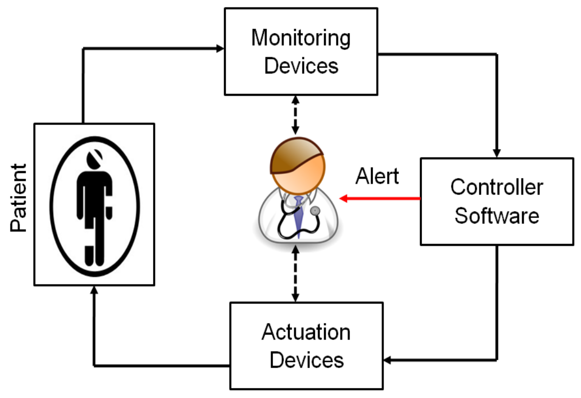


Fig. 1: The Generic View of MCPS

Moreover, the recent advances in wireless sensor networks (WSN), medical sensors, and cloud computing are making CPS as a potential candidate for advanced health care applications including in-hospital and in-home patient care which is known as a Medical Cyber Physical Systems (MCPS) [3]. These advances promise to provide CPS the ability to observe patient conditions remotely and take actions regardless of the patient's location.

In MCPS, traditional clinical scenarios are viewed as closed-loop systems in which the caregivers are the controllers, medical devices act as sensors and actuators, and patients are the physical plants as shown in Figure 1. This system incorporates feedback from the plant, and the software is said to be a closed-loop controller. Examples of closed-loop controller software in MCPS include automated insulin pumps, pacemakers, anesthesia devices, etc. The patient's physiological conditions are the inputs to the software and the output is the action to take, such as how much medicine to deliver. The software maintains state (e.g. the values of past readings) and then awaits another input of the patient's condition before deciding on the next action, and thus the loop continues [4].

Thus, MCPS are life safety critical, context-aware, and networked systems of medical devices that are collectively involved in treating a patient. These systems are planned to use in hospitals to provide high-quality continuous care for patients in complex clinical scenarios. The goal of MCPS is to improve the effectiveness of patient care by providing personalized treatment through sensing for ensuring safety. However, the increased scope and complexity of MCPS relative to traditional medical systems present numerous developmental challenges [5]. In addition, the quality assurance of controller software is one of the utmost importance because the cost (either monetary or in terms of health and lives) of faults in the implementations can be enormous. In medical systems, a single information flaw can lead to unrecoverable damages, hence ensuring validation of MCPS is a must. Thus, to ensure the safety of patients, the whole systems must be carefully tested before being applied to the reality. To make a MCPS effective, it is necessary to test all possible paths of the system state. Testing all the paths provide full coverage for ensuring patient's safety to anticipate the identification of all errors.

Various techniques for ensuring safety or validating safe work flow of medical systems have been proposed in literature. Most of the techniques represent the full system and system interaction in model based manner using modeling tools or various self proposed model notations. These techniques focus on model based testing approaches by simulating generated state models of the system. However, these model based approaches do not provide any test scenarios for logically testing the transitions of whole system states. As all possible logical transitions of the system states are not tested during model simulations, hence the untested transitions can cause severe safety threats while real life patient medication. To overcome these limitations, automated all possible test scenario generation is required which can ensure safety.

Lee et. al. states MCPS as a safety critical system where safety introduces numerous challenges, such as achieving high assurance in system software, intoperability, context-aware intelligence, autonomy, security and privacy, and device verifiability etc [5]. Andrew et. al. proposed a model-based framework for testing MCPS that includes a modeling language with formal semantics considering MCPS as a Virtual Medial Device (VMD) [6]. They also used a Medical Application Platform (MAP) that provides the necessary deployment support for the VMD models [6]. However, the technique only gets notification about an occurrence of a failure, it cannot detect the actual cause of the failure. Wu et. al. presented a work flow adaptation and validation protocol to help physicians safely adapt work flows which can react to patient adverse events based on the pathophysiological models [7]. Their technique focused on dynamically adapting the work flow and validating safety requirements. This technique requires continuous physician involvement and supports only single work flow at a time. Moreover, the safety validation protocol does not produce all possible test scenarios for the transitions of the whole system. Banerjee et. al. proposed a theoretical framework for testing cyber physical interactions, empowering CPS researchers to systematically design solutions for ensuring safety, security, or sustainability [8]. Nevertheless, the framework focus on considering the spatio-temporal physical environment while testing the interactions among physical aspects, computing aspects and their tight coupled interactions only.

In the paper [9], the author proposed an automatic test case generation approach for closed loop controller software. Their proposed approach takes source code as input and generates test cases to cover all possible paths of the source code. This approach consists of two steps: test case generation and test suite reduction. The test case generation step generates test cases using KLEE, a test case generation tool. KLEE uses symbolic execution to identify all possible sub paths, and constraint solver to generate test cases that satisfy those paths. However KLEE generates redundant test cases that cover the same path or state repeatedly. Therefore, the test suite reduction step generates a sub-path coverage matrix from the KLEE generated test cases. It then applies Minimum Set Cover Problem on the coverage matrix and finds the reduced test suite that covers

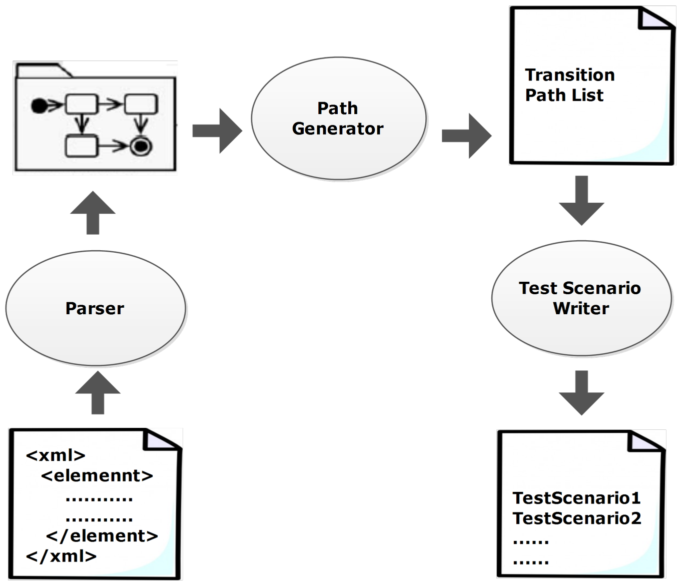


Fig. 2: Proposed Automatic Test Scenario Generation Approach

all sub-paths. The approach seems a source based test generation approach. However, system source code is often inconsistent with system requirements. Therefore, the approach lacks semantic information which is found in system UML models, such as state diagram. As a result, the generated test cases fail to cover all the states of system models.

None of the above mentioned technique tries to extract all possible transitions among the various states of the system. Since MCPS is a highly sensitive application of CPS and its tiny information, device or system flaw can cause serious damage of a patient, thus, it is necessary to test all possible paths of the system to ensure patient safety. Therefore, only all the possible transitions among the states of the system state model can ensure the safety of MCPS.

This paper proposes an automatic test scenario generation procedure for MCPS that can cover all possible transitions. The technique first takes the system state diagrams as input in a XML format. The formatted XMLs are parsed to extract system states, transitions, conditions and event triggers. Simple Depth First Search (DFS) algorithm is applied on the extracted state diagrams to extract all possible simple unique paths from the diagrams. Based on these extracted transition paths test scenarios are prepared and stored in a text document. As all the possible interactions are considered, hence these test scenarios would be able to ensure safety of the system.

The rest of the paper is organized as follows. Section 2 describes the proposed methodology about how all possible transition scenarios can be produced. Section 3 describes an case study of generic patient controlled analgesia pump model as an example of MCPS. Section 4 discusses the probable improvements of the proposed technique and Section 5 concludes the paper.

2. PROPOSED AUTOMATIC TEST SCENARIO GENERATION APPROACH

In this section, an automated test scenario generation approach for MCPS is proposed. As mentioned above, the existing techniques only focus on representing the system using modeling notations and model simulations for safety checking. However, these tech-

niques are unable to ensure the safety of a patient because they do not consider all possible transition sequences of the state models, which are required to be tested for safety assurance. To overcome the limitations of above mentioned techniques, an automated test scenario generation technique for MCPS systems is proposed.
The overview of the proposed technique has been illustrated in Figure 2. To deploy the proposed approach, an realistic assumption is made that the state diagram of the system is available, since state diagram is produced in the requirement specification phase. The proposed automated test scenario generation approach consists three modules - *Parser*, *Path Generator* and *Test Scenario Writer*. Each of these modules perform predefined responsibilities. *Parser* module processes input data into program readable format. It takes UML state diagram as input, and extracts necessary information for test generation. The *Path Generator* module generates all possible paths required to be tested from the extracted UML information. Finally, the *Test Scenario Writer* module produces and stores test cases for those extracted paths in order to assist validation.

2.1 Parser

The *Parser* module is the input data provider of the proposed approach. It receives XML formatted state diagrams as input. Each state diagram is generally associated to a class. As a result, the transition paths in a state diagram represent the valid paths in the system class which needs to be tested. Therefore, this module extracts information about states, transitions, corresponding guards and action events from the tags of the UML state diagrams. Once these information are extracted from the XML files, they are used to generate state model graph for further processing, where the graph nodes represent system states, and edges represent transitions as well as events and conditions.

2.2 Path Generator

The *Path Generator* module is responsible for generating possible transition paths for test scenario generation. In order to ensure safety of a system, all possible paths in the system execution needs to be tested. If all the states and transitions are covered with test scenarios, it ensures that all possible paths of the system is tested. Therefore this module receives the extracted state graph from the *Parser* module as input. It then applies simple DFS algorithm on the state diagrams to generate all possible transition paths of the system. However, this step can produce a number of redundant paths. Thus, path generation module considers simple paths which do not contain same edge twice. The consideration of simple path enables the proposed approach to handle self loops and extract unique transition paths also. These generated paths are stored in a document and act as input of the next module for test scenario construction.

2.3 Test Scenario Writer

The *Test Scenario Writer* performs the task of producing test cases for covering possible transitions in the state diagrams. Once all the possible transitions among the states have been extracted, this module takes the transition path lists as input. In order to generate test scenario for each extracted transition path Algorithm 1 is proposed. Algorithm 1 takes a path list (P) and a corresponding state diagram (S) to which the path list belongs to as inputs. Each transition in a state diagram contains a unique id. So, A path list, P contains sequential unique ids (*id*) of state diagram transitions which constitute to the path. In return, the algorithm generates a list of test scenarios (*testList*). To represent test scenarios, a complex data structure, *Test* is declared (line 2). Each test scenario has two attributes - test scenario id and a transition sequence to be tested. To generate test scenario for each listed path a *for* loop is declared (line 4). For each path, p in the inputted path list, an instance of type *Test* is initialized (line 5). The algorithm then assigns a unique number to the *id* attribute of the test scenario instance (line 6). A inner for loop is declared for iterating through the transitions of each path, p (line 7). In the next step, the algorithm call a function *getTransition* to extract a transition from a corresponding state diagram (line 8). The function takes an id and a state diagram as arguments. It returns the corresponding transition having the specified id from the state diagram. This transition is a complex data type, *Transition* which has two attributes - *transitionId* and *label*. Basically this *label* attribute represents the method call, guard condition or action event responsible for the transition. In the next step, the algorithm adds the transition label to the transition sequence of the test scenario (*test*) as the next transition to be tested (line 9). Finally, the algorithm adds the test scenario to the final list of test scenarios (line 11).
All the generated test scenarios are written and stored in text documents. The generated test scenarios assist the testers with test paths that needs to be covered to ensure the safety of the system.

Algorithm 1 Test Scenario Generation

```
Input: A list of possible transition paths ($P$) and a corresponding
    state diagram (S)
Output: A list of scenarios ($testList$)
 1: Begin
 2: $Transition\ t, Test\ test$
 3: $List < Test > testList$
 4: for each $p \in P$ do
 5:     $test \leftarrow$ new Test()
 6:     $test.id \leftarrow p + 1$
 7:     for each $transitionId \in p$ do
 8:         $t \leftarrow getTransition(transitionId, S)$
 9:         $test.transitionSequence.add(t.label)$
10:     end for
11:     $testList.add(test)$
12: end for
13: End
```

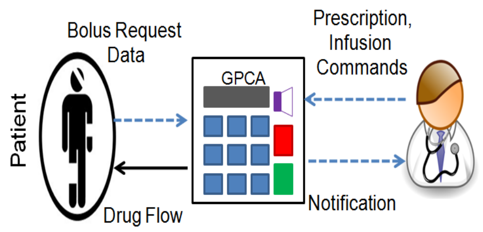


Fig. 4: MCPS with GPCA

3. CASE STUDY

A complex Generic Patient Controlled Analgesia (GPCA)] [10], [11] Infusion Pump is considered here as an example of MCPS to illustrate the proposed approach. The test scenario generation of GPCA can serve as a generic example to generate the test scenarios for testing other close loop systems also. GPCA Infusion system is a MCPS system that allows patients to a self-administer for a

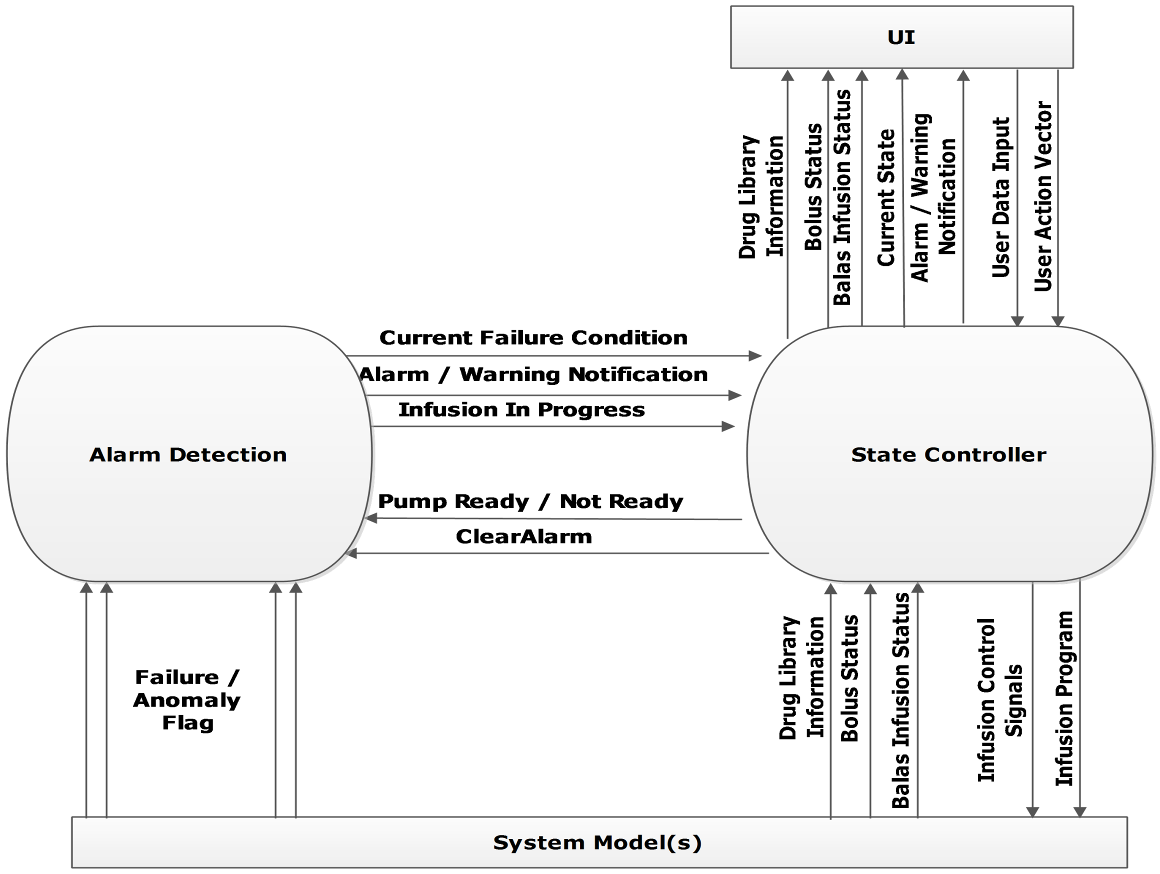


Fig. 3: The GPCA Model System Architecture [11]

controlled amount of drug, by patient bolus mode to alleviate acute pain. There may be multiple bolus modes. In clinician bolus mode, the drug is delivered at an elevated rate in response to a clinician's request. For example, the clinician may prescribe an elevated rate of infusion for a period of time at the beginning of infusion therapy. Figure 4 shows a GPCA device in a typical usage environment, such as a hospital or a clinic.

The GPCA devices are usually built with capability to monitor and notify the clinician of exceptional conditions, for example, if the drug reservoir is running low or if there are air bubbles in the system. The GPCA has three primary functions: (1) deliver the drug based on the prescribed schedule and patient requests, (2) prevent hazards that may arise during its usage, and (3) monitor and notify the clinician of certain exceptional conditions encountered. Thus GPCA can be act as both monitoring and actuation device. Before going to the details of the test scenario generation of GPCA, the overview of GPCA and state diagram of GPCA is presented for better understanding.

3.1 Overview of GPCA Architecture

An architectural overview of the GPCA model has been illustrated in Figure 3 [11]. The GPCA consists of three tiers - System Component Tier, actual GPCA Pump Tier and User Interface Tier. At the lowest tier in this architecture are system model components representing actual pump components (such as the pump controller, delivery mechanism, power unit, etc.). The middle tier consists of the core GPCA model, implemented as a pair of communicating state machines (labeled State Controller and Alarm Detecting Component in the Figure 3). The top tier represents the pump's User Interface (UI) that is used to display various messages, and allows users to program the pump. Signals provided by the core GPCA model enable communication with the user interface and various system components. In this paper, the generated test cases for the middle GPCA tier (State Controller and Alarm Detecting Component) is presented for page limitation.

The primary purpose of the GPCA State Controller is to model the drug administration process such as the programmed basal rate and any bolus doses that the patient may request. Based on these drug transformation, the Alarm Detecting Component identifies any inconsistencies or system states and notifies the State Controller.

3.1.1 The GPCA State Model Representation.

A state model represents the states of the system along with transition based on the events or conditions which cause state transitions. Similar to general state models, the GPCA model also consists a set of state machines denoting the various states of the infusion pump and the transitions between these states. The transitions are triggered by various events or guard conditions. The events represent user queries and guard conditions represent mathematical or Boolean expressions consisting of state variables. Hence, the

GPCA state models are generally represented by five components. These components are Initial state, Set of possible states, Set of transitions such as events and guard conditions, Set of final states and Set of actions performed on state variables as a result of transitions.

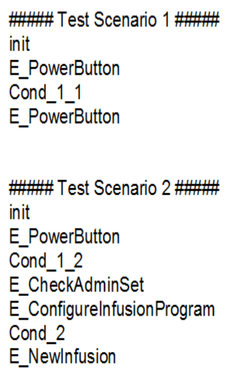
The overall state model diagram of the *State Controller* is presented in Figure 5 [11] which is used as a input of the proposed approach. The state model is further divided into sub machines for two reasons. Firstly, states and transitions that collectively perform a specific task should be isolated into one group. Secondly, once a sub-machine is revised or elaborated, the rest of the state machine remains unchanged, except for its interaction with the sub-machine. The *Check Drug Routine*, *Infusion Configuration Routine* and *Infusion Session Sub-Machine* are the sub modules of GPCA state controller. Each of the sub modules can be further expressed into the state diagram and need to be tested. As an example state diagram of *Infusion Session Sub-Machine* is shown in Figure 6. For the page limitation, other sub modules are left abstract in this paper. The *Infusion Session Sub-Machine* module represents infusion operations of the pump. The *Infusion Configuration Routine* demonstrates the work flow that a user needs to go through in order to prescribe a correct infusion treatment for the patient. On the other hand, the *Check Drug Routine* sub module checks whether the loaded drug satisfies with the prescription defined in the drug library. The *Alarm Detection* module receives signals from sensors and sends them back to *State Controller* module and vice versa. In this paper, test case generation for overall GPCA sate machine (Figure 5) and infusion pump's software(InfusionSession SubMachine module (Figure 6)) is presented that controls the drug infusion and raises alarms to notify the clinician when hazards are detected.

3.2 Test Case Generation for GPCA

At first, the proposed approach receives the XML formatted state diagrams as input. The state diagrams of GPCA system are converted into XML format using Enterprise Architect [12]. A partial view of the XML formatted GPCA state model is illustrated in Figure 7. The *Parser* module then parses the input state diagram to extract necessary information. For example, it identifies the states and transitions by analyzing the <state> and <transition> tags respectively. Similarly, the information regarding method invocation, guard conditions and action events are extracted from the attributes of <transition> tag.

The *Path Generator* module uses the extracted UML information to generate state graphs and applies DFS algorithm on these graphs to produce all possible simple paths as stated in subsection 2.2. The lists of generated paths are then sent to the *Test Scenario Writer* module for further processing.

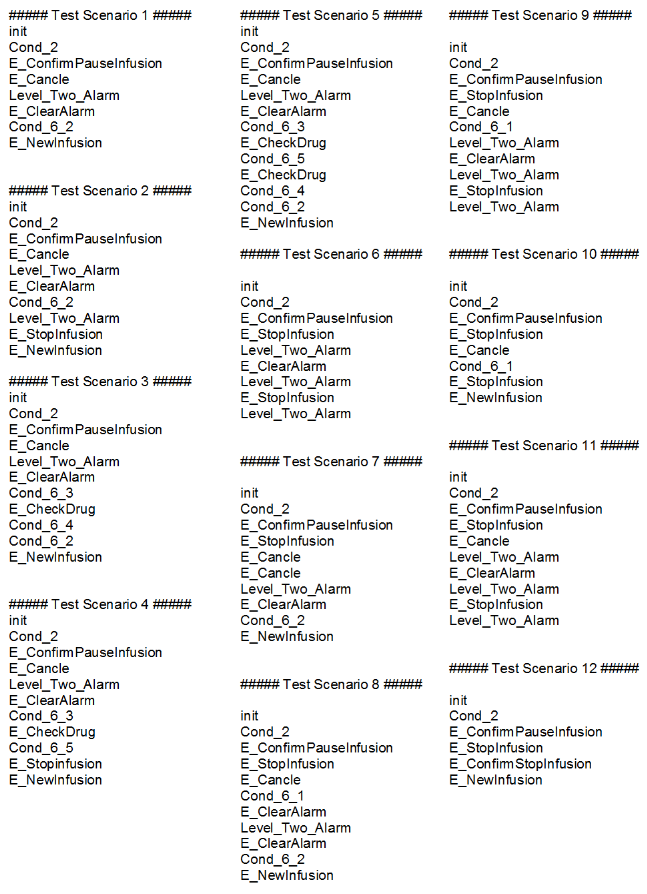
Finally, *Test Scenario Writer* module applies Algorithm 1 on the extracted path list to generate test scenarios for the possible state interactions of the GPCA. The number of generated possible test scenarios for *GPCA State Controller* state model of Figure 5 and *Infusion Session Sub Machine* state model of Figure 6, are 2 and 12 respectively. Generated test scenarios for these models are showed in Figure 8 and 9 respectively. A sample scenario of the *GPCA State Controller* model of Figure 6 depicts that the system can get activated from *PowerOff* state, and can create a new infusion instruction after checking few configuration status. This scenario is executed by visiting several state transitions such as - *PowerOff–> Post–> PostDone–> CheckDrugRoutine–> InfusionConfigurationRoutine–> InfusionSessionSubMachine–>CheckDrugRoutine*. To ensure the safety of the system this sample scenario needs to be tested.



```
##### Test Scenario 1 #####
init
E_PowerButton
Cond_1_1
E_PowerButton

##### Test Scenario 2 #####
init
E_PowerButton
Cond_1_2
E_CheckAdminSet
E_ConfigureInfusionProgram
Cond_2
E_NewInfusion
```

Fig. 8: Generated Test Scenario of GPCA State Controller



```
##### Test Scenario 1 #####
init
Cond_2
E_ConfirmPauseInfusion
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Cond_6_2
E_NewInfusion

##### Test Scenario 2 #####
init
Cond_2
E_ConfirmPauseInfusion
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Cond_6_2
Level_Two_Alarm
E_StopInfusion
E_NewInfusion

##### Test Scenario 3 #####
init
Cond_2
E_ConfirmPauseInfusion
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Cond_6_3
E_CheckDrug
Cond_6_4
Cond_6_2
E_NewInfusion

##### Test Scenario 4 #####
init
Cond_2
E_ConfirmPauseInfusion
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Cond_6_3
E_CheckDrug
Cond_6_5
E_Stopinfusion
E_NewInfusion

##### Test Scenario 5 #####
init
Cond_2
E_ConfirmPauseInfusion
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Cond_6_3
E_CheckDrug
Cond_6_5
E_CheckDrug
Cond_6_4
Cond_6_2
E_NewInfusion

##### Test Scenario 6 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
Level_Two_Alarm
E_ClearAlarm
Level_Two_Alarm
E_StopInfusion
Level_Two_Alarm

##### Test Scenario 7 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
E_Cancle
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Cond_6_2
E_NewInfusion

##### Test Scenario 8 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
E_Cancle
Cond_6_1
E_ClearAlarm
Level_Two_Alarm
E_ClearAlarm
Cond_6_2
E_NewInfusion

##### Test Scenario 9 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
E_Cancle
Cond_6_1
Level_Two_Alarm
E_ClearAlarm
Level_Two_Alarm
E_StopInfusion
Level_Two_Alarm

##### Test Scenario 10 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
E_Cancle
Cond_6_1
E_StopInfusion
E_NewInfusion

##### Test Scenario 11 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
E_Cancle
Level_Two_Alarm
E_ClearAlarm
Level_Two_Alarm
E_StopInfusion
Level_Two_Alarm

##### Test Scenario 12 #####

init
Cond_2
E_ConfirmPauseInfusion
E_StopInfusion
E_ConfirmStopInfusion
E_NewInfusion
```

Fig. 9: Generated Test Scenario of Infusion Session Sub Machine

4. DISCUSSION

The proposed technique assists the task of tester by automatically generating test scenarios of GPCA by using the state transitions. In this paper, generated test scenarios for GPCA state controller module and InfussionSessionSubMachine sub module is depicted. However, the test cases of other modules are not provided for page

limitations. By using this proposed technique, the whole system as a module, as well as all sub modules and the interaction among different modules can be tested. In addition, the proposed technique can be applicable for the test case generation of any CPS applications. However, initially only simple paths of the state models is considered, as a result test scenarios regarding self loop transitions are ignored.
However, gradual modification and application of the proposed technique on different context projects would enhance the technique and increase its effectiveness.

5. CONCLUSION

The significant expansion of safety-critical systems like MCPS results in a qualitatively larger space of behaviors that needs to be ”covered” during testing, not only at the system level but also at subsystem and unit levels to ensure the safety of a patient. Thus, to handle this kind of large system, automation in test case generation is necessary. A major challenge in this area is to automatically generate a set of test cases that, collectively, guarantees safety of a patient and system. To ensure the patient safety, an automated test scenario generation approach for MCPS by generating all possible test cases is proposed in this paper.
The proposed approach uses UML model such as state diagram as a input which has emerged as the de facto standard for modeling software systems also [13]. The overall implementation is performed using three steps. The first step is *Parser*, which takes the XML formatted state diagrams as input and extracts necessary information for graph generation. The *Path Generator* step takes the generated graph as input and uses DFS algorithm to extract all possible transition paths. The final step, *Test Scenario Writer* generates test scenarios based on the extracted transitions paths.
For analyzing the applicability, the proposed technique has been applied on an open source GPCA Model. The technique has been successful in generating test scenarios which covers all possible transitions of the model.
However, the proposed test case generation approach is based on semantic from XMLs and ignores syntax from source. In future, it is needed to incorporate syntax source code also for more concrete test case generation. In addition, further investigation is required to find the optimal set of test cases with full coverage.

6. REFERENCES

[1] Lee, Edward A. “Cyber physical systems: Design challenges.” Object oriented real-time distributed computing (isorc), 2008 11th IEEE international symposium on. IEEE, 2008.

[2] Nower, Naushin, Yasuo Tan, and Azman Osman Lim. “Traffic pattern based data recovery scheme for cyber-physical systems.” IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 97.9 (2014): 1926-1936.

[3] Lee, Insup, and Oleg Sokolsky. “Medical cyber physical systems.” Design Automation Conference (DAC), 2010 47th ACM/IEEE. IEEE, 2010.

[4] Murugesan, Anitha, et al. “From requirements to code: model based development of a medical cyber physical system.” (2014).

[5] Lee, Insup, et al. “Challenges and research directions in medical cyberphysical systems.” Proceedings of the IEEE 100.1 (2012): 75-90.

[6] King, Andrew L., et al. “Assuring the safety of on-demand medical cyber-physical systems.” Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on. IEEE, 2013.

[7] Wu, Po-Liang, et al. “Safe Workflow Adaptation and Validation Protocol for Medical Cyber-Physical Systems.” Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on. IEEE, 2015.

[8] Banerjee, Ayan, et al. “Ensuring safety, security, and sustainability of mission-critical cyberphysical systems.” Proceedings of the IEEE 100.1 (2012): 283-299.

[9] Murphy, Christian, Zoher Zoomkawalla, and Koichiro Narita. “Automatic test case generation and test suite reduction for closed-loop controller software.” (2013).

[10] Generic PCA Infusion Pump Reference Implementation, `https://rtg.cis.upenn.edu/medical/gpca/gpca.html`, 23 12 2011.

[11] The generic patient controlled analgesia pump model `https://rtg.cis.upenn.edu/gip-UPenn/gip-docs/GPCA%20Pump%20Model.doc`

[12] Enterprise Architect, `http://www.sparxsystems.com/`, Online; accessed 19 August 2016.

[13] Sarma, Monalisa, and Rajib Mall. “Automatic test case generation from UML models.” Information Technology,(ICIT 2007). 10th International Conference on. IEEE, 2007.

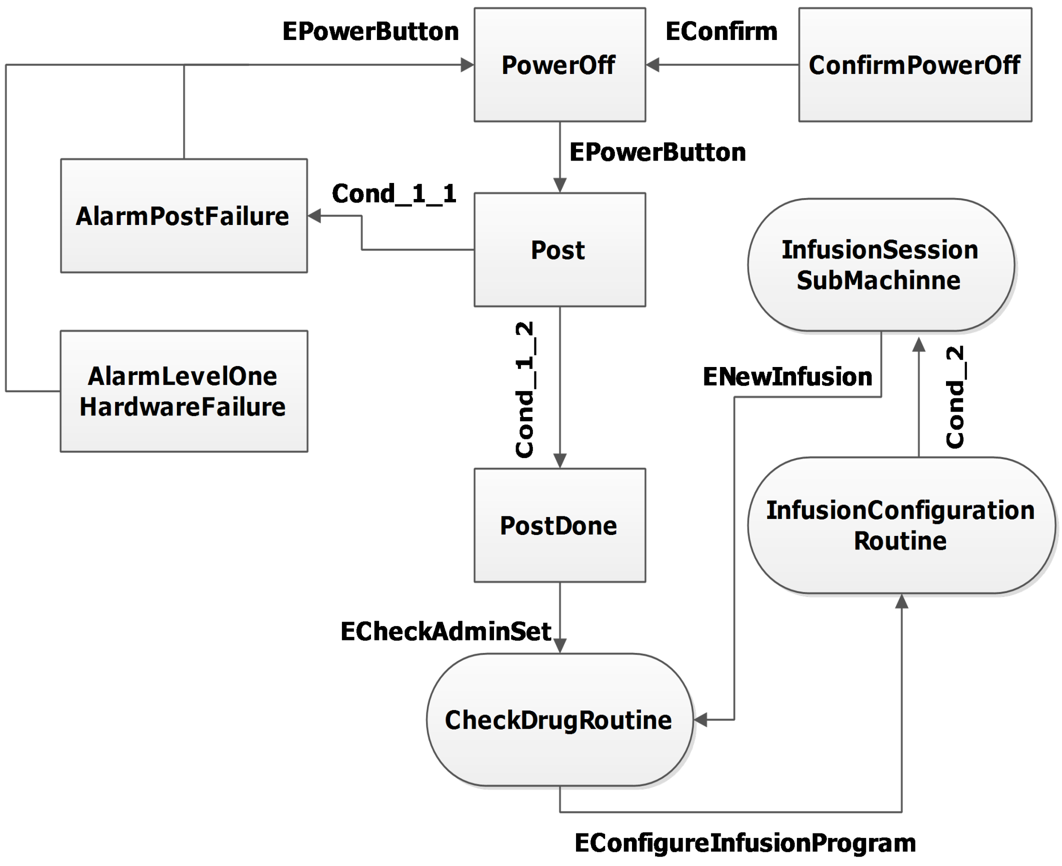


Fig. 5: The State Diagram of GPCA State Controller [11]

]

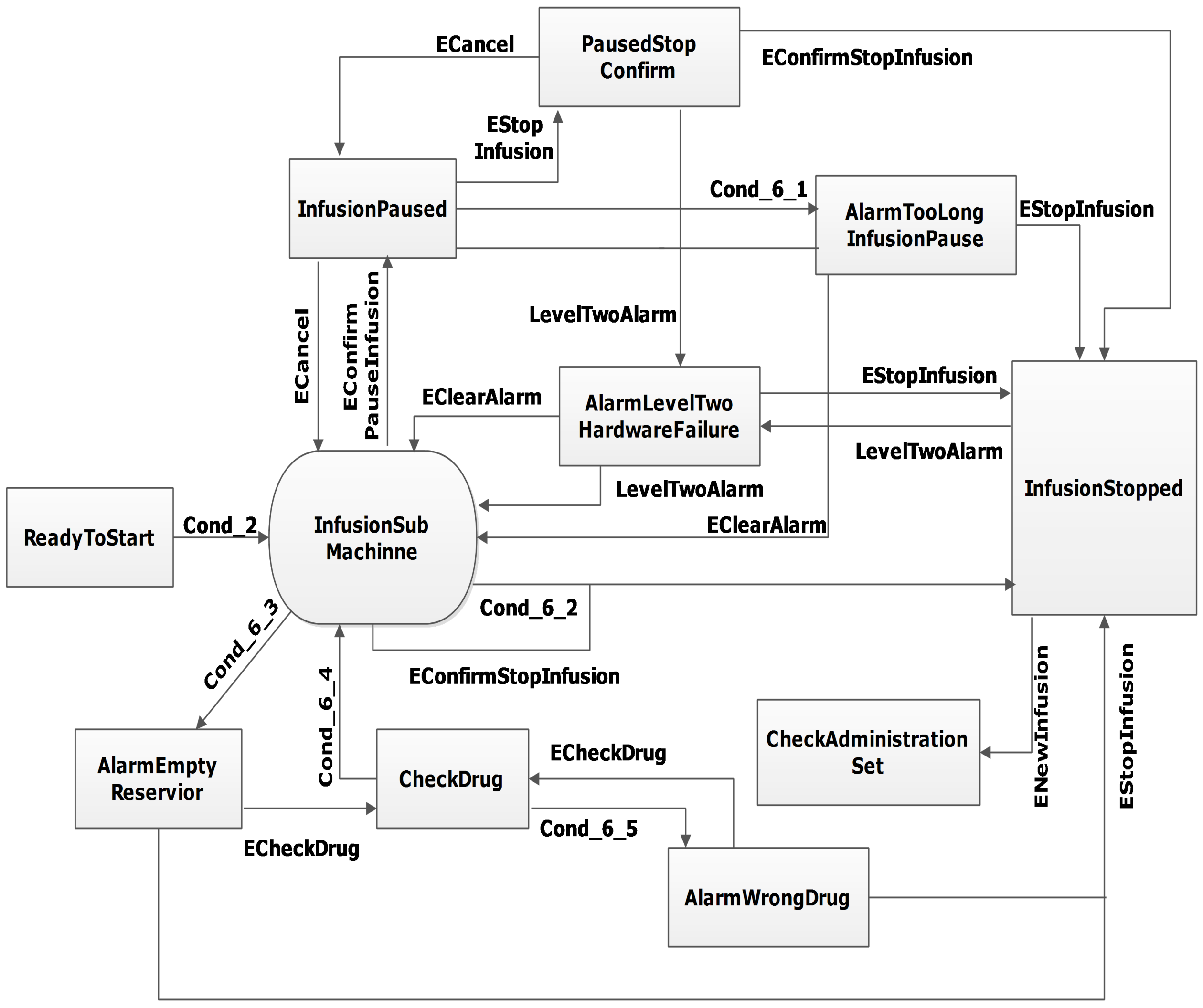


Fig. 6: The State Digaram of Infusion Session Sub-Machine [11]

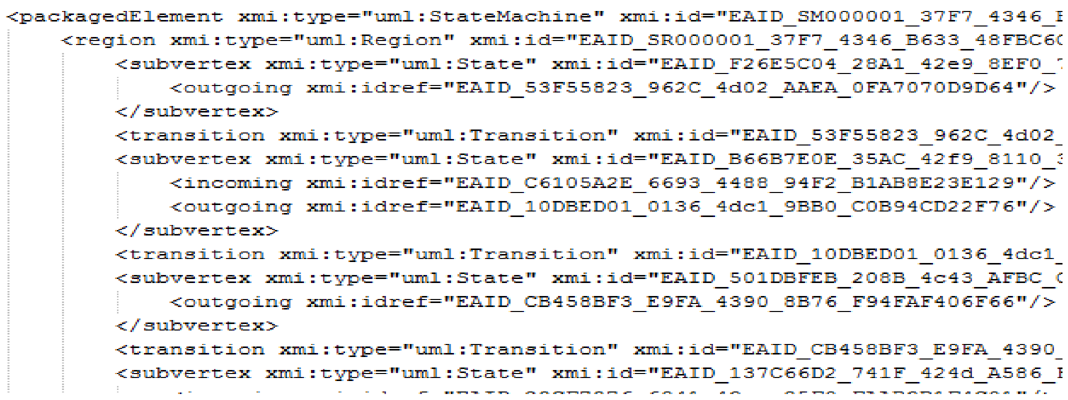

```
<packagedElement xmi:type="uml:StateMachine" xmi:id="EAID_SM000001_37F7_4346_
    <region xmi:type="uml:Region" xmi:id="EAID_SR000001_37F7_4346_B633_48FBC6
        <subvertex xmi:type="uml:State" xmi:id="EAID_F26E5C04_28A1_42e9_8EF0_
            <outgoing xmi:idref="EAID_53F55823_962C_4d02_AAEA_0FA7070D9D64"/>
        </subvertex>
        <transition xmi:type="uml:Transition" xmi:id="EAID_53F55823_962C_4d02_
        <subvertex xmi:type="uml:State" xmi:id="EAID_B66B7E0E_35AC_42f9_8110_
            <incoming xmi:idref="EAID_C6105A2E_6693_4488_94F2_B1AB8E23E129"/>
            <outgoing xmi:idref="EAID_10DBED01_0136_4dc1_9BB0_C0B94CD22F76"/>
        </subvertex>
        <transition xmi:type="uml:Transition" xmi:id="EAID_10DBED01_0136_4dc1_
        <subvertex xmi:type="uml:State" xmi:id="EAID_501DBFEB_208B_4c43_AFBC_
            <outgoing xmi:idref="EAID_CB458BF3_E9FA_4390_8B76_F94FAF406F66"/>
        </subvertex>
        <transition xmi:type="uml:Transition" xmi:id="EAID_CB458BF3_E9FA_4390_
        <subvertex xmi:type="uml:State" xmi:id="EAID_137C66D2_741F_424d_A586_
```

Fig. 7: XML view of GPCA State Model(Partial)