fSysML: Foundational Executable SysML for Cyber-Physical System Modeling

Omar Badreddin[1], Vahdat Abdelzad[2], Timothy C. Lethbridge[3], Maged Elaasar[4]

[1] University of Texas, El Paso El Paso, Texas, U.S.A
obbadreddin@UTEP.edu
[2] EECS, University of Ottawa, Ottawa, Ontario, Canada
v.abdelzad@uottawa.ca
[3] EECS, University of Ottawa, Ottawa, Ontario, Canada
tcl@eecs.uottawa.ca
[4]Modelware Solutions, La Canada Flintridge, CA 91011, USA
melaasar@gmail.com

Abstract. System engineers are heavy users of modeling and design languages such as SysML. These design languages enable them to design, refine, verify, and test systems early in development. On the other hand, and especially with the emergence of agile methodologies, design and development activities in software engineering are intermingled and performed in iterations. Modern systems, however, exhibit increasing interdependence between software and physical components. Hence, there is a growing need to develop design languages that can bridge the gap between system and software engineering communities. This paper proposes fSysML, a foundational and executable subset of SysML geared towards facilitating the development of modern Cyber-Physical Systems. fSysML defines both a surface syntax for a SysML subset, and an executable semantics that is directly mapped to a modern object-oriented language. fSysML is demonstrated by the development of a self-adaptive system from the healthcare domain.

Keywords: SysML · UML · Cyber-Physical Systems · Self-Adaptive Systems · Textual Language.

1 Introduction

Cyber-Physical Systems (CPS) are integrations of computation, networking, and physical processes [1]. Software and networks monitor and control the physical processes, with feedback loops where physical processes affect computations and vice versa [2]. One of the key characteristics of such systems is their close integration and interdependence of both software and hardware components.

Software and hardware components are typically developed following different development processes and methodologies. One key distinction is propensity to change. In the software world, change is embraced and can influence many aspects of development. While in the physical world, it is costly and cannot be accommodated without significant cost.

As a result, software and systems engineers have different perspectives regarding design and development activities. System engineers adopt modeling and design whole-heartedly, and do not typically require modeling languages to be executable. In fact, execution for a systems engineer often refers to execution of a simulation or model-based testing. System engineers rely on precise and elaborate models to test and verify systems before the commencement of development activity. Moreover, design models are the gold standard against which any development artifact needs to be tested.

Software engineers, on the other hand, utilize more integrated approaches and work iteratively and incrementally. Modern agile software development processes encourage the delivery of an executable artifact at every iteration. These partially complete software systems are used to discover additional requirements, and feed into planning, scheduling and staffing management.

With the emergence of CPSs, there is an emerging need for platform and design languages that can accommodate both system and software development processes. In this paper, we address this emerging need by introducing fSysML; foundational SysML for CPSs. fSysML defines a textual surface language for a subset of SysML and integrates this subset with an executable subset of UML. The result is a language that can define many aspects of CPS properties, including Blocks, Requirements, Goals, Users, Use Cases, as well as behavioral and compositional aspects. We demonstrate fSysML using the design of a self-adaptive system from the healthcare domain. Self-adaptive systems demonstrate further interdependencies between both software and physical components, making it ideal for the demonstration of fSysML.

This paper is organized as follows. We introduce the motivation and significance of this research in Section 2. A background on self-adaptive systems, system modeling and MDE is introduced in Section 3. Section 4 introduces a running example followed by detailed explanation of fSysML in Section 5. The language grammar and related work are covered in Section 6 and 7 respectively. Finally, we conclude with a discussion of future work and conclusion.

2 Motivation and Significance

The authors of this paper are actively collaborating with a national aerospace agency and are investigating a roadmap for adoption of Model Driven Engineering (MDE) paradigm. From early stages of the investigation, the researchers observed a significant discrepancy between system and software engineers. The bulk of the development effort in the system engineering side is closely related to development of various types of design models. Key approvals, certifications, and testing are performed against system models. Software is treated as a black-box component with elaborate behavioral specification. The software engineers use design models sparingly and typically target development of an executable artifact.

Systems engineers reported on their need for a textual syntax, equivalent to the visual representation, of their SysML models. The rationale is that text can be versioned and merged more effectively along with the software artifacts. Furthermore, it maybe easier to manipulate and layout textual artifacts than visual models, particular-

ly as models become large and complex. More importantly, such a textual language will facilitate collaboration and integration between system and software engineers. This observation has motivated the research presented in this paper; namely, the development of a textual SysML language that can enable effective collaboration between both software and system engineers.

2.1 Significance

There is a recognized conflict between MDE and agile methodologies [12]. MDE promotes upfront designs, where models become the key development artifacts. Agile, on the other hand, promotes shorter development cycles by focusing on delivering executable partial systems. This conflict becomes even more prominent in the development of CPSs. Solving this conflict has the potential to significantly improve the development practices of CPSs. More importantly, it can help resolve a long-standing challenge in the software engineering community; namely, the limited adoption of MDE practices [13].

There is significant evidence that software engineers do not in fact adopt UML and modeling as much as desired [1, 6]. Studies of software engineers in the wild suggest multitude of limitations with MDE methodologies including: 1) challenges with versioning and merging of models [5] 2) limitations with model inter-changeability and portability [6] 3) inadequacy of MDE in development of small systems [7], 4) lack of adequate support for model based collaboration [8].

3 Background

In this section, we introduce background on system and software modeling, and self-Adaptive systems.

3.1 Model Driven Software Engineering

The Object Management Group (OMG) has adopted a vision where models become the key development artifacts, from which executable elements can be generated. The premise includes improved productivity and quality of software systems. UML has emerged as the de facto modeling notation in software engineering and has been well evaluated academically.

Since UML is a general-purpose modeling notation, not all of its elements have well-defined execution semantics. This has resulted in numerous challenges for the development of precise models. As a result, OMG has defined a subset of UML with defined semantics called Foundational UML (fUML) [9]. Action Language for Foundational UML (ALF) is an executable language that defines actions for fUML [10]. ALF has been designed to look like modern object-oriented languages to facilitate adoption by software engineers. fUML and ALF's emergence has in part informed and influenced the development of fSysML.

3.2 System Modeling

The landscape of system modeling is more complex. System engineers use a wider variety of modeling notations and design languages. System designs are used not only to test and verify the system, but also to derive the cost and schedule associated with its development. In this paper, we use SysML as the representative system design language, but the approach proposed in this paper can be applicable beyond SysML.

SysML borrows many notations from UML; this includes use case modeling, structural modeling, and behavioral modeling using a variant of UML state machines. SysML adds additional design notation to model requirements, blocks and components, and system parameters.

3.3 Self-Adaptive Systems

Self–adaptive systems have the unique property of dynamically adapting to changes in environment, operating context, or changes in system goals or priorities [18]. Self-adaptation strategies are typically implemented at the architecture level by identifying adaptation strategies and mechanisms for dynamic applications of such strategies at run time [17]. Whether a specific strategy is effective or not is typically measured by measuring the quantified outcome or performance of the entire system. Such quantitative evaluation is typically performed at runtime [19].

Self-adaptation is realized by identifying and implementing a number of adaptation strategies or tactics. Tactics application is performed in response to an external change in the operating environment or deterioration in system performance, and aims at improving one or more of the system's goals. Take for example a web server that is supposed to service user requests. The system's goals may include speedy responses, and to keep the number of time-out requests to a minimum. At peak hours when performance declines, the web-server may apply a performance tactic such as load sharing with a back-up server.

4 Running Example

To demonstrate fSysML, we use a self-adaptive monitoring system from the healthcare domain. The monitoring system reports on patients key vital signs in real time and notifies specific caregivers when certain conditions are present. The overall goal of the system is to speed patient discharge from the Heart Unit and reduce per-patient costs; while at the same time maintaining a low re-admission rate and high patient satisfactions (Appendix – Goal Model).

Amongst the monitored vital signs, the heart rate and oxygen level are measured by embedded sensors. These sensors produce a continuous stream of data. The system analyzes the data on the fly and stores data points only when certain rules succeed. The system behavior is driven in part by readings from these sensors (Appendix 1– Behavior Model). The next section elaborates on this example further by highlighting key language elements.

5 fSysML Overview

fSysML targets system and software engineers in an integrated manner. As such, it supports system and software modeling with little or no distinction. In the following sections we will illustrate the language elements focusing on system modeling.

5.1 System Goals and Objectives

The language supports the definition of key system goals. A goal can be measured by physical elements (such as a sensor) or by a combination of elements that can be quantified by a Key Performance Indicator (KPI). Goals can be broken down into sub-goals. In this case, two goals can both contribute to a higher-level goal through an "AND" or an "OR" contribution. "And" contribution means the lower satisfaction of the sub-goals is transferred to the super goal, while OR contribution means the higher satisfaction is transferred to the super goal. Contributions may also be associated with a weight. Listing 1 is an fSysML snippet that describes the system goals. A visual depiction of System Goal is illustrated in the Appendix 1.

```
goal Discharged {
  contributesTo PatientSatisfaction{0.7};
  stakeholder Patient, Physician; }

goal NormalHeartRate  {
  contributesTo Discharge{0.5};
  stakeholder Patient, Physician;
  KPI heartRateMeasurement threshold = 50;
  failureLimit = 0;
  correctiveAction = heartRateTreatment;  }

goal NormalBloodOxygenLevel  {
  contributesTo Discharge{0.5};
  stakeholder Patient, Physician;
  KPI bloodOxygenMeasurement threshold = 25;
  failureLimit = 0;
  correctiveAction = oxygenTreatment; }

goal LowCost {
 contributesTo PatientSatisfaction{0.3};
 stakeholder Patient, Physician; }

softGoal PatientSatisfaction  {
  stakeholder Patient, Physician;
  decompositionType = 'and'; }
```

Listing 1. System Goals and Goal Compositions

5.2 System Failure and Failure Tolerance

Not all system goals need to succeed at all times. For example, if an oxygen-supplying system is not connected to a patient, then oxygen concentration levels need not be above a threshold. Some goals can fail sometimes without repercussions, and some goals cannot fail at all. This is defined by modeling the failure limit. A failure

limit of one means that if a goal fails once, an adaptive action must be taken (adaptive actions are discussed later). Similarly, a failure limit of two means that a goal can fail twice over a specified period of time without repercussions (i.e, associated adaptive action will be applied after the second goal failure occurrence). A failure limit of zero means that a specific goal should not be allowed to fail at all. In such a case, the system must apply rules to take precautionary actions if the goal trends towards failure.

5.3 Adaptive Actions

A goal failure, even if permitted, may or may not trigger an adaptive action. An adaptive action is defined by a set of instructions (or tactics) with the aim of eliminating or reducing the reoccurrence of goal failure. The effectiveness of a particular adaptive action is measured by 1) the satisfaction of the goal immediately connected to the adaptive action element and 2) the top-level system goal satisfaction. Adaptive actions are defined algorithmically using ALF.

5.4 Scenarios (Processes) and Tasks

A scenario or a process is a sequence of tasks that can take place in a defined sequence. The sequence can be dependent on conditions or actions that can occur at run time. fSysML supports the definition of scenarios, join and fork control flows. As shown in the following example (Listing 2), the scenario Triage contributes to the goal Discharge. In this scenario, if a treatment is required, then the path along the treatment task is taken. Otherwise, the path along discharge task is taken.

```
task Treat {
  actor Physician;
  dependsOn TreatmentForm; }

task Discharge {
  actor Physician;
  dependsOn DischargeForm; }

task OxygenTreatment {
  actor Physician; }

task HeartRateTreatment {
  actor Physician;
  form TreatementForm; }

scenario Triage {
  satisfies Discharged;
  [needsTreatment]?
  {Treat}:{Discharge}; }

form DischargeForm  {
  date mandatory;
  time optional;
  Patient.firstName mandatory;
  Patient.lastName mandatory;
  Physician.firstName mandatory;
  Physician.lastName mandatory;
  ..  }

form TreatmentForm {
  date mandatory;
  time mandatory;
  Patient.firstName mandatory;
  Patient.lastName mandatory;
  Patient.diagnosis mandatory;
  Patient.diagnosisCode mandatory;
  Physician.firstName mandatory;
  Physician.lastName mandatory;
  Physician.employeeID mandatory;}
```

Listing 2. Tasks and Scenarios

5.5 Behavioral Modeling

fSysML defines a textual syntax for UML state machines to define system behavior. The state machine subset supported by fSysML includes states, events, guards, transitions, actions, entry, do, and exit activities.

At the top level of the state machine in Listing 3, the system is functioning under one of two states, Normal and Abnormal. In both states, the streams of sensor data is received and analyzed. If data streams show abnormalities, the system continues to monitor but is now functioning under the abnormal state. While in the Abnormal state, the system may itself trigger some corrective actions to try to remedy the deficiencies. If successful, the system may return to the Normal state. These behavioral elements specify the behavior of some system components or Blocks (discussed in the next subsection). In this example, we illustrate the behavioral modeling for the monitor control unit.

The state machine syntax is an extension of Umple's state machine modeling syntax [15]. Umple is a model oriented UML Action Language. We extended Umple's syntax, rather than extending ALF's syntax for two reasons; 1) ALF's state machine syntax is under development and has not been standardized to date, and 2) ALF's syntax is declarative (the syntax declares states and transitions in a linear fashion). We found Umple's syntax to have better comprehension based on the feedback from our system engineer collaborators.

```
 // From BDD :
block ControlUnit {
..
..
  block HR_Sensor {
    // behavioral definition
    state Normal {
      region HR_Sensing {
        NormalRate {
          entry/{ updateDisplay();}
          do{monitor_HR();
            analyze_HR_Rule();}
        Abnormal_HR_Detected->Abnormal_HR;}
       }
      region Dormant { .. }
     }
     state Abnormal { .. }  } }
```

Listing 3. Behavioral Model

```
block HR_Sensor {
  parts:
    protective_housing,
    mount_assembly,
    sensor_module,
    electronics_assembly,
    display
  values:
    dimension: Size
    power: Width
    field_of_sensing: int
    orientation: int
  flow_ports:
    in_ligh_in: Light
    sensor_IO: Sensor_interface
  standard_ports:
   control: Sensor_signal }
```

Listing 4. Block Definition

5.6 System Block Definitions and Component Modeling

Block Definition Diagrams (BDD) are an important part of system modeling [11]. fSysML models physical aspects of the system following SysML specifications, but in a way to facilitate its integration with other aspects of software modeling. Listing 4 illustrates a block definition for the Hear Rate Sensor. Listing 5 illustrates the composition of the Monitoring Unit, of which the Heart rate Sensor is a part of.

A block has four groups of properties; namely, parts, values, flow ports and standard ports. The Block Definition Diagram (BDD) describes composition relationships

denoted by --- (Listing 5). For example, HR_Sensor is composed of three External Wirings, one enclosure and one battery. The syntax for blocks and block definitions is different than the rest of fSysML syntax. For example, dimension is of type size. One would expect that the type would precede the name of the attribute such as follows:

```
values {
  size dimension; }
```

This choice of syntax may in fact look unusual for a software engineer. This choice in fSysML was influenced by 1) SysML visual diagrams, and 2) system engineers' preferences.

5.7 Users, Actors and User Groups

fSysML defines users of the systems, and user groups that can share a common characteristics, such as access privilege. A specific instance of a user is referred to as an Actor. Users, Actors and User Groups can participate in a scenario, and may be assigned to tasks. Users and Actors need not be a human actor, but can be any other subsystem or a block.

```
bdd MonitoringComponent {
  bdd HR_Sensor {
    1 --- 3 External_Wiring;
    1 --- 1 Enclosure;
    1 --- 1 Battery; }
  bdd  Oxygen_Sensor { .. }
  ..
  ..
  bdd Enclosure {
    1 --- 1 front_housing;
    1 --- 1 back_housing;
    1 --- 2 secure_strap;
    ..
    ..
  }  }
```

Listing 5. Block Definition Modeling

```
actor Patient {
  int patientID;
  string firstName;
  string lastName;
  int diagnosisCode; }

actor CareProvider; {
  int current_heart_rate;
  int target_heart_rate;
  int current_blood_oxygen;
  int target_blood_oxygen; }

actor Physician {
  int employeeID;
  .. }
```

Listing 6. System Actors

6 Language Grammar

fSysML is developed using Xtext [16], a platform for DSL development. The grammar is defined using a BNF-like syntax. Listing 7 illustrates key languages elements, some of which has been discussed in previous sections. Here, we briefly discuss the elements that have not been presented in the previous sections.

The language supports tagging model elements. Tagged values appear between curly brackets and may have typed data elements. Tagging is useful when engineers want to analyze a subset of the system elements. For example, a subset of system elements may be selected for regression testing, or for measuring overhead costs. A Soft Goal is similar to a Goal, except that a Soft Goal typically refers to a non-functional aspect of the system. Nevertheless, a Soft Goal may still be quantified using a KPI. A scenario may itself contain other scenarios. fSysML treats a task as a unitary action that cannot be further broken down.

```
  fSysML:
  elements+=(Actor|Usergroup|KPI|Goal|Softgoal|stateMachine|Block|BDD)*;

  Contribution:
  contributesTo' (goals += [Goal]|goals += [Softgoal])'{'weight = INT'};';

   TaggedValue:
   'taggedValue' '=' '{'(ID '=' Datatype',')*(ID '=' Datatype)'}';

   KPI:
   'KPI' name=ID'{'(TaggedValue)?(goals_contributed_to +=Contribution)*'};';

   Goal:
   'Goal' name = ID '{'(TaggedValue)? (goals_contributed_to += Contribution)*
('Stakeholder' actors += [Actor]';')+ (('KPI' indicators += [KPI]';')* |
('SoftGoal' softgoals += [Softgoal]';')*)('decompositionType' '='
dType=DecompositionType';')? '};' ;

   Softgoal:
   'SoftGoal' name = ID '{'(TaggedValue)? (goals_contributed_to += Contribu-
tion)+ ('Stakeholder' actors += [Actor]';')+ (('KPI' indicators += [KPI]';')*
| ('SoftGoal' softgoals += [Softgoal]';')*) '};';

   Task:
   'Task' name = ID '{'(TaggedValue)? 'Actor' actor += [Actor]';' ('Form' forms
+= [Form]';')* '}' ;

   Scenario:
   'Scenario' name = ID '{'('satisfy' satisfied_goals += [Goal])+('depends' de-
pendent_forms += [Form])+ (tasks += [Task]'->')+ ('['boolean'],
{'ss_one+=SubScenario'},{'ss_two+=SubScenario'}'|(tasks += [Task]';') ;

   SubScenario:
   (tasks += [Task]'->')+ ('['boolean'],{'ss_one+=SubScenario'},
{'ss_two+=SubScenario'}'|(tasks += [Task]';') ;
```

Listing 7. Part of fSysML Grammar

7 Related Work

There has been numerous and growing trend in supporting textual UML modeling. Such approaches have materialized in a number of tools. textUML supports a library and an API for creating UML diagrams using Java syntax [22]. tUML is another tool that supports both visual and textual modeling of UML [23]. tUML allows engineers to mix-in code elements along with the textual UML models. Key goal of tUML is to facilitate the quick and easy manipulation of sketchy UML models. PlantUML is a textual and verbose modeling tool that focuses on flexibility [24]. Actions in PlantUML supports unstructured textual elements that do not comply with a meta-model.

fSysML extents the standard SysML language with goal modeling. This is used to help assess adaptive actions effectiveness. Laleau et al [21] have proposed an approach to combine SysML with requirement modeling that includes extending SysML with Goal Modeling. Their approach entails formal specifications of Goals using B language [20]. Vahdat et al [25] have proposed a textual Goal Modeling using the GRL standard [26]. Vahdat's approach follows GRL standard strictly to facilitate

transformations between the textual and visual manifestation. fSysML goal models are designed specifically to support self-adaptation by quantifying adaptive actions at various levels of goals hierarchy.

Manzoor [27] and Derler [31] proposed a Domain Specific Language (DSL) for modeling of self-adaptive System. Their approach treats self-adaptation as a special type of requirements (requirements under uncertainty). The proposed DSL integrates elements of RELAX [29], a requirement engineering language for self-adaptive systems, into the proposed DSL. Recognizing the heterogeneity in cyber-physical and self-adaptive systems, Heinzemann et al [28] have proposed a development process that spans the multiple disciplines involved in the development of such systems. The proposed process focuses on the integration control engineering and software engineering. fSysML is well suited for such integrated processes.

8 Future Work and Conclusion

Cyber-Physical Systems exhibit greater interplay between both physical elements and software elements, with significant feedback loops and shared controls. Development of such systems requires both software and system engineers to collaborate. Where modern software engineers follow an iterative and incremental processes, system engineers adopt disciplined design-oriented processes.

To facilitate the development of Cyber-Physical Systems, we propose a design language that encompass the following key properties. 1) Enables the design and modeling of both physical and software elements. This is achieved by defining a subset of SysML and specifying a surface textual syntax for that subset. The result is integrated with a textual syntax for modeling of software elements. 2) Eliminates to the greatest extent possible the distinction between physical and software elements. This is achieved by defining a uniform language for both types of elements.

8.1 Parametric and Constraints modeling

Parametric modeling definition in the language is ongoing. Parametric modeling for systems supports trade-off analysis, model based testing, and typically supports the definition and execution of system level constraints. We are investigating the use of a language such as OCL [14] to help define constraints and parametric aspects of system modeling. These investigations include introducing an OCL or an OCL-like syntax as a unit test in fSysML. This requires that parameters that are used in parametric modeling must be defined in fSysML structural models, or as part of Block Definition modeling.

8.2 Environment, Risks, and Uncertainty

CBS function in an environment exhibiting continuous properties with many sources of uncertainties. How to effectively model risks and uncertainties and present it as integral elements in the language is yet to be investigated. Effective modeling of these elements contributes to addressing key state-of-the-art research challenges, in-

cluding 1) model testing of cyber-physical systems under uncertainties [32], 2) model based analytics of CBS [33], among others.

8.3 System Requirements Modeling

System requirements is another area of ongoing investigations. In SysML, requirements are defined in a hierarchical fashion. A requirement may have attributes such as an ID and text, and may be related to one or more modeling elements. To implement this in fSysML, we must introduce new concepts, such as satisfies (or isSatisfiedBy), contributesTo (or contributedToBy), and realize (or realizedBy). These concepts will relate requirement elements to various system and software model elements. Semantics of such concepts must have implications to testing and analysis activities. For example, a change of a model element should be reflected to all related elements in requirements. Unit testing should cover all requirements contribute to relationships and other modeling elements.

References

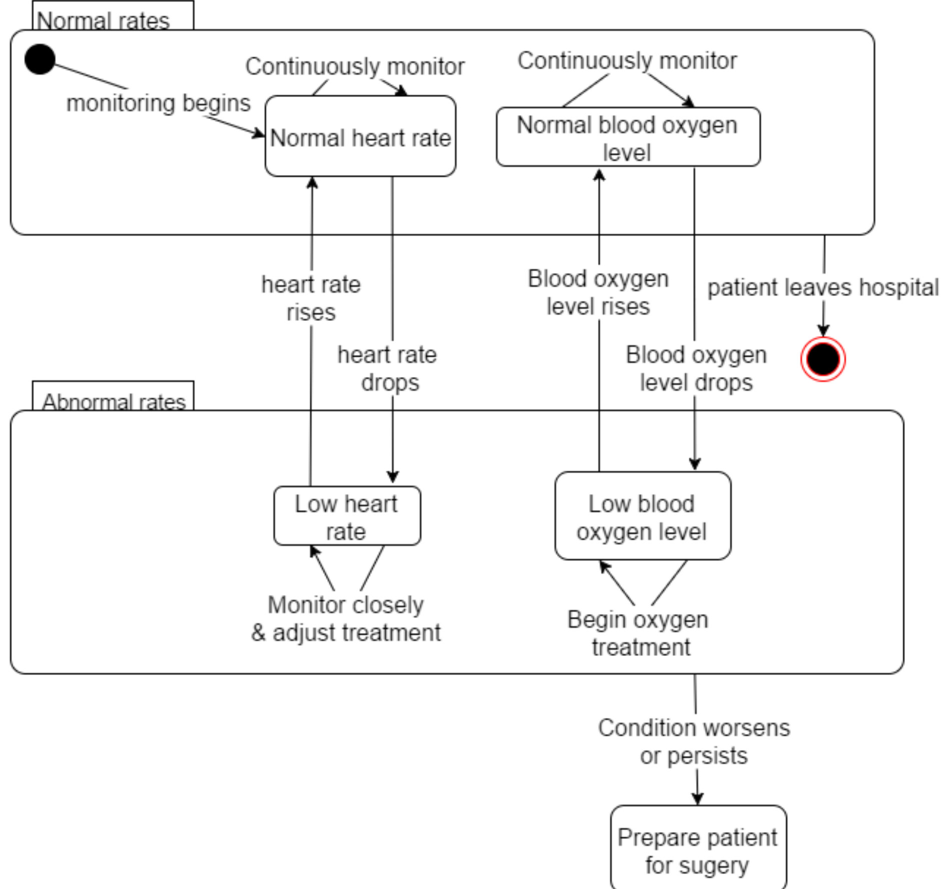
1. Shi, Jianhua, et al. "A survey of cyber-physical systems." Wireless Communications and Signal Processing (WCSP), 2011 International Conference on. IEEE, 2011.
2. Lee, Edward A., and Sanjit A. Seshia. "Introduction to Embedded Systems, A cyber physical approach". First Edition, (2014).
3. Mohagheghi, Parastoo, et al. "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases." Empirical Software Engineering 18.1 (2013): 89-116.
4. Petre, Marian. "“No shit” or “Oh, shit!”: responses to observations on the use of UML in professional practice." Software & Systems Modeling 13.4 (2014): 1225-1235.
5. Badreddin, Omar, Timothy C. Lethbridge, and Andrew Forward. "A novel approach to versioning and merging model and code uniformly." 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014. IEEE, 2014.
6. Huang, Shihong, Vaishali Gohel, and Sam Hsu. "Towards interoperability of UML tools for exchanging high-fidelity diagrams." 25th Annual ACM international Conference on Design of Communication. ACM, 2007.
7. Stevens, Perdita. "Small-scale XMI programming: a revolution in UML tool use?" Automated Software Engineering 10.1 (2003): 7-21.
8. Dobing, Brian, and Jeffrey Parsons. "How UML is used." Communications of the ACM 49.5 (2006): 109-113.
9. Lazar, Codrut Lucian, et al. "Using a fUML Action Language to construct UML models." Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009 11th International Symposium on. IEEE, 2009.
10. Badreddin, Omar, Timothy C. Lethbridge, and Andrew Forward. "Investigation and evaluation of UML Action Languages." 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014 . IEEE, 2014.
11. Huang, Edward, Randeep Ramamurthy, and Leon F. McGinnis. "System and simulation modeling using SysML." 39th conference on Winter simulation: 40 years! The best is yet to come. IEEE Press, 2007.

12. Hailpern, Brent, and Peri Tarr. "Model-driven development: The good, the bad, and the ugly." IBM systems journal 45.3 (2006): 451.
13. Forward, Andrew, Omar Badreddin, and Timothy C. Lethbridge. "Perceptions of software modeling: a survey of software practitioners." 5th workshop from code centric to model centric: evaluating the effectiveness of MDD (C2M: EEMDD). 2010.
14. Warmer, Jos B., and Anneke G. Kleppe. "The Object Constraint Language: Precise Modeling With UML (Addison-Wesley Object Technology Series)." (1998).
15. Badreddin, Omar. "Umple: a model-oriented programming language." Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 2010.
16. Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, 2010.
17. Oreizy, Peyman, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. "An architecture-based approach to self-adaptive software." IEEE Intelligent systems 14, no. 3 (1999): 54-62.
18. De Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Litoiu, M., ... & Weyns, D. (2013). Software engineering for self-adaptive systems: A second research roadmap. In Software Engineering for Self-Adaptive Systems II (pp. 1-32). Springer Berlin Heidelberg.
19. Aziz B, Arenas A, Bicarregui J, Ponsard C, Massonet P (2009) From goal-oriented requirements to are Event-B specifications. In: First Nasa formal method symposium (NFM 2009), Moffett Field, California, USA.
20. Calinescu, R., Ghezzi, C., Kwiatkowska, M., & Mirandola, R. (2012). Self-adaptive software needs quantitative verification at runtime. Communications of the ACM, 55(9), 69-77.
21. Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., & Tatibouet, B. (2010). A first attempt to combine SysML requirements diagrams and B. Innovations in Systems and Software Engineering, 6(1-2), 47-54.
22. Chaves, R. 2009. "TextUML". http://abstratt.github.io/textuml/readme.html
23. Jouault, Frédéric, and Jérôme Delatour. "Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Sys-tems." OCL@ MoDELS. 2014.
24. PlantUML modeling tool. Available online: http://plantuml.com/.
25. Abdelzad, Vahdat, Daniel Amyot, and Timothy C. Lethbridge. "Adding a Textual Syntax to an Existing Graphical Modeling Language: Experience Report with GRL." International SDL Forum. Springer International Publishing, 2015.
26. Heaven, William, and Emmanuel Letier. "Simulating and optimising design decisions in quantitative goal models." 2011 IEEE 19th International Requirements Engineering Conference. IEEE, 2011.
27. Whittle, Jon, et al. "RELAX: a language to address uncertainty in self-adaptive systems requirement." Requirements Engineering 15.2 (2010): 177-196.
28. Heinzemann, C., Sudmann, O., Schäfer, W., & Tichy, M. (2013, May). A discipline-spanning development process for self-adaptive mechatronic systems. In Proceedings of the 2013 International Conference on Software and System Process (pp. 36-45). ACM.
29. Ahmad, Manzoor. "First step towards a domain specific language for self-adaptive systems." In 2010 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE), pp. 285-290. IEEE, 2010.

30. Whittle, Jon, et al. "RELAX: a language to address uncertainty in self-adaptive systems requirement." Requirements Engineering 15.2 (2010): 177-196.
31. Derler, Patricia, Edward A. Lee, and Alberto Sangiovanni Vincentelli. "Modeling cyber–physical systems." *Proceedings of the IEEE* 100.1 (2012): 13-28.
32. Briand, Lionel, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. "Testing the untestable: model testing of complex software-intensive systems." In Proceedings of the 38th International Conference on Software Engineering Companion, pp. 789-792. ACM, 2016.
33. Sharma, Abhishek B., Franjo Ivančić, Alexandru Niculescu-Mizil, Haifeng Chen, and Guofei Jiang. "Modeling and analytics for cyber-physical systems in the age of big data." ACM SIGMETRICS Performance Evaluation Review 41, no. 4 (2014): 74-77.

APPENDICES

Appendix 1: Behavioral Model



Appendix 2: Goal Model