Measuring Maintainability of DPRA Models: A Pragmatic Approach

Irina Rychkova[1], Fabrice Boissier[1], Hassane Chraibi[2], and Valentin Rychkov[2]

[1] Université Paris 1 Panthéon-Sorbonne,
12, Place du Panthéon, 75005 Paris, France
[2] EDF R&D, EDF Lab Paris-Saclay,
7 Bd. Gaspart Monge, Palaiseau, France

Abstract. Dynamic Probabilistic Risk Assessment (DPRA) is a powerful concept that is used to evaluate design and safety of complex industrial systems. A DPRA model uses a conceptual system representation as a formal basis for simulation and analysis. In this paper we consider an adaptive maintenance of DPRA models that consist in modifying and extending a simplified model to a real-size DPRA model. We propose an approach for quantitative maintainability assessment of DPRA models created with an industrial modeling tool called PyCATSHOO. We review and adopt some metrics from conceptual modeling, software engineering and OO design for assessing maintainability of PyCATSHOO models. On the example of well-known "Heated Room" test case, we illustrate how the selected metrics can serve as early indicators of model modifiability and complexity. These indicators would allow experts to make better decisions early in the DPRA model development life cycle.

Keywords: maintainability metrics, conceptual models, object oriented design, DPRA models

1 Introduction

Dynamic Probabilistic Risk Assessment is a powerful concept that is used to evaluate design and safety of complex systems where the static reliability methods like fault trees find their limits [2]. A DPRA model is grounded on a conceptual representation of a system: it formally describes some aspects of the physical world (for example, a complex mechanical system) for purposes of understanding and communication [24]; it serves a formal basis for further system simulation and analysis. For feasibility, proof of concept, algorithm benchmarking and other preliminary studies simplified DPRA models are used. Adaptive maintenance is an important part of a DPRA model life cycle: it consists in modifying and extending a simplified model to a real-size DPRA model.

In this work, we propose a maintainability assessment approach for DPRA models created with an industrial modeling tool called PyCATSHOO[6,7]. PyCATSHOO is a tool dedicated for dependability analysis of hybrid systems developed and used by EDF. PyCATSHOO models are executable modules that can be written in Python or C++ and interpreted by PyCATSHOO engine.

We define and test a set of metrics that can serve as early indicators of PyCATSHOO model modifiability and complexity. We review some well-known metrics from conceptual modeling, software engineering and object-oriented design, including size measures, complexity measures, lexical measures and OO-specific measures [9,20,13,26,5]. Based on our review, we select and adapt a set of metrics applicable to DPRA models and PyCATSHOO models in particular.

We make an assumption that *the selected metrics can show a difference between two PyCATSHOO model designs already at the early phase of model development life cycle*, helping experts to make better decisions.

To validate this assumption, we apply the selected metrics and assess two designs of the Heated Room system. "Heated Room" is a well-known test case reported in [3,6]. It describes a system that consists of a room and a heater that can switch on and off to maintain the ambient temperature. This example illustrates a hybrid system that combines deterministic continuous phenomena (i.e., temperature evolution) and stochastic discrete behaviour (i.e., functioning of a heater).

We create two sets of PyCATSHOO models as illustrated in Table 1: Set 1 follows the original design ideas from [6], Set 2 represents an alternative model design promoting the low coupling design principle. The corresponding models in two sets are semantically equivalent, i.e., they demonstrate the same simulation traces. We compare measurements for the model sets and discuss the results.

The reminder of this paper is organised as follows: Section 2 presents DPRA models and PyCATSHOO modeling tool; Section 3 discusses the related works on maintainability assessment; Section 4 presents our approach for maintainability assessment of PyCATSHOO models; In Section 5 we describe two alternative designs of the PyCATSHOO models for the Heated Room test case. We assess maintainability of these designs and discuss our results in Section 6; Section 7 presents our conclusions.

Table 1: PyCATSHOO models of the Heated Room test case

Model description:	Set 1 Design 1	Set 2 Design 2
Initial model: 1 heater H + 1 room R	Case 0	Case 0a
System level modificaiton: 4 independent heaters H0..H3 + 1 room	Case 1	Case 1a
Component level modification: standby redundancy of heaters H0 - main; H1..H3 - backups	Case 2	Case 2a

2   DPRA Models and PyCATSHOO Modeling Tool

EDF has a long-standing experience in using and developing DPRA tools for complex systems. PyCATSHOO is one of the tools developed over last few decades at EDF R&D. PyCATSHOO implements the concept of Piecewise Deterministic Markov Process (PDMP) using distributed stochastic hybrid automata. The principles of this paradigm are described in details in [6].

PyCATSHOO models are used for advanced risk assessment of EDF's hydro and nuclear electrical generation fleet. PyCATSHOO is grounded on the Object-Oriented (OO) and Multi-Agent System (MAS) paradigms[22]. Following the OO paradigm, PyCATSHOO defines a system, its subsystem or component as a class - an abstract entity that can be instantiated into objects. The latter are concrete entities that communicate by message passing and that are able to perform actions on their own encapsulated states. This paradigm has been successfully implemented for modeling and analysis of stochastic hybrid systems as reported in [21]. Following MAS, PyCATSHOO models a system as a collection of objects with a reactive agent-like behavior. A reactive agent acts using a stimulus-response mode where only its current state and its perception of its environment are relevant.

2.1 DPRA Modeling with PyCATSHOO

PyCATSHOO offers a flexible modeling framework that allows for defining generic components (classes) of hybrid stochastic automata to model a given system or a class of systems with a particular behaviour. A hybrid stochastic automaton may exhibit random transitions between its states according to a predefined probability law. It may also exhibit deterministic transitions governed by the evolution of physical parameters.

A modeling process with PyCATSHOO can be summarised as follows:

– Conceptual level: A system is decomposed into elementary subsystems, components.
– Component level: Each system component is described with a set of hybrid stochastic automata, state variables and message boxes. Message boxes ensure message exchange between components.
– System level: To define the system, the components are instantiated from their corresponding classes. Components' message boxes are connected according to the system topology.

A DPRA model in PyCATSHOO combines the characteristics of conceptual model and a software application: it formally describes some aspects of the physical world for purposes of understanding and communication [24]; it serves a formal basis for further system simulation and analysis.

PyCATSHOO offers Application Programming Interfaces (APIs) in Python and C++ languages. Once the model is designed, the system behaviour is simulated. An analyst needs to use Monte Carlo sampling if the system exhibits random transitions. Sequences (time histories of the system evolution) that lead to desirable end states are traced and clustered.

In [2], various modeling tools for DPRA are discussed. Whereas some modeling tools propose a visual modeling interface, model complexity, high development and maintenance costs are considered the main obstacle for efficient use of DPRA models in industry [8]. Quantitative measures of model maintainability would be of a great value, helping the experts to make better decisions early in the DPRA model development life cycle.

3 Maintainability of Models: State of the Art

ISO 9000 is a set of international standards on quality management. It defines quality as *"the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs"* [15]. Maintainability is a quality characteristic that is defined as *"a set of attributes that bear on the effort needed to make specified modifications."*

3.1 Maintainability in Software Engineering

In Standard Glossary of Software Engineering Terminology [16] software maintainability is defined as "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment".

According to ISO/IEC 25010, maintainability is a sub-characteristic of product quality that can be associated with more concrete, "measurable", quality metrics. Various types of metrics accepted in SE include: size metrics (e.g., Line of Code), lexical metrics (e.g., Halstead software science metrics [13]), metrics based on control flow graph (e.g., Mc'Cabe's cyclomatic complexity [20]) etc.

Metrics specific to Object-Oriented paradigm focus on OO concepts such as object, class, attribute, inheritance, method and message passing. Chidamber & Kemerer's OO metrics [5] are among the most successful predictors in SE. They include metrics focused on object coupling. In [18] ten software metrics and their impact on the maintainability are studied. In [27], a systematic review of software maintainability prediction models and metrics is presented. According to this review, a list of successful software maintainability predictors include Halstead metrics[13], McCabe's complexity[20] and size metrics.

Abreu's Metrics for Object-Oriented Design (MOOD) are presented in [1] and evaluated in [14]. According to MOOD, various mechanisms like encapsulation, inheritance, coupling and polymorphism can influence reliability or maintainability of software.

The maintainability index (MI) is a compound metric [26] that helps to determine how easy it will be to maintain a particular body of code. MI uses the Halstead Volume, Cyclomatic Complexity, Total source lines of code.

The models and metrics above address the maintainability at later phases of software development life cycle. In the next part of this section, we consider maintainability at design phase. In particular, maintainability of conceptual models.

3.2 Maintainability in Conceptual Modeling

Conceptual modeling is the activity of formally describing some aspects of the physical and social world around us for the purposes of understanding and communication [24]. While ISO/IEC 25010 family standards is widely accepted for evaluating software systems, no equivalent standard for evaluating quality of conceptual models exist.

In [23,25,4] frameworks for conceptual modeling quality are presented. In [25], the empirical quality is considered as a good indicator of the maintainability. In [4] maintainability of conceptual schema is defined as *"the ease with which the conceptual schema can evolve"*. Quantitative analysis and estimation of conceptual model quality remains challenging due to lack of measurement [23].

An important body of knowledge is developed adopting and extending the metrics from software engineering to the conceptual modeling. These metrics are used to estimate quality of conceptual models and UML diagrams in particular. In [11], a survey of metrics for UML class diagrams is presented. In [19], a suite of metrics for UML use case diagrams and complexity of UML class diagrams is proposed. Directly measurable metrics are used as an early estimate of development efforts, implementation time and cost of the system under development. In [9], a set of metrics to measure structural complexity of UML class diagram are proposed and validated. The authors promote an idea that the structural properties (such as structural complexity and size) of a UML class diagram have an effect on its modifiability and maintainability. In [10], the same group of researchers proposes metrics for measuring complexity of UML statechart diagrams.

In [28], 'Maintainability Estimation Model for Object-Oriented software in Design phase' (MEMOOD) is presented. This model estimates the maintainability of class diagrams in terms of their understandability and modifiability. Modifiability is evaluated using the number of classes, generalisations, aggregation hierarchies, generalization hierarchies, direct inheritance tree.

4 Maintainability Assessment of PyCATSHOO Models

Intrinsically, maintainability is associated with the maintenance process, which represents the majority of the costs of a software development life-cycle [27]. It is valid for the model development life cycle as well.

Assessment of conceptual model maintainability can help designers to anticipate model complexity, to incorporate required enhancements and to improve consequently the maintainability of the final software [28]. In this work we adapt and apply several metrics from SE and OO design to evaluate the maintainability of PyCATSHOO models early at the model development life cycle.

Adapting a maintainability definition from [16], we define maintainability of a DPRA model as *the ease with which a model or its component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment*.

According to [17], maintainability consists of five sub-characteristics: modularity, reusability, analyzability, modifiability and testability. *Modifiability* sub-characteristic is most relevant for DPRA models; it specifies a degree to which a product (a DPRA model in our case) can be effectively and efficiently modified without introducing defects.

Similar to software system maintainability, various types of maintainance for DPRA models can be identified:

- Adaptive - modifying the model to cope with changes in the environment;

- Perfective – improving or enhancing a model to improve overall performance;
- Corrective – diagnosing and fixing errors, possibly ones found by users;
- Preventive – increasing maintainability or reliability to prevent problems in the future (i.e., model architecture, design).

In this work, we focus on *adaptive maintenance* that reflects a transformation of simplified DPRA models to real-size models.

4.1 Adaptive Maintainability in PyCATSHOO Models

Different classes of modifications can be introduced into a PyCATSHOO model while adapting it to a real-size model. In this work, we consider two classes of PyCATSHOO model modifications:

1. Component level modifications - modifications that consist in adapting structure and/or behavior of a model component (e.g., state variables, PDMP equation methods for continuous variables, start/stop conditions for PDMP controller, transition conditions, message boxes etc.).
2. System level modifications - modifications that consist in adapting structure and topology of the system (e.g., number of component instances, their parameters, dependencies, connections via message boxes etc.).

Each modification class can be related to different requirements and consequently different technical solutions [12]. We argue that the "right" architectural and design decisions made for a simplified DPRA model pay off, improving maintainability and reducing complexity of the real-size DPRA model and PyCATSHOO models in particular. The maintainability metrics can serve as indicators for DPRA domain experts in order to assess their design and architectural decisions early in the modeling.

4.2 Selecting Maintainability Metrics

After a review of some existing metrics focusing on maintainability, we select the set of metrics applicable to DPRA models and PyCATSHOO models in particular. We propose a new metric to measure relative modifications - RLOC. We summarize the selected metrics in Table 2. Our choice was purely pragmatic: possibility of seamless integration of metrics into current modeling process with PyCATSHOO and availability of measurement tools were the main criteria for our metrics selection. Radon[3] is a Python tool which computes various code metrics. Radon supports raw metrics, Cyclomatic Complexity, Halstead metrics and the Maintainability Index. Cloc[4] - Count Lines of Code - is a tool that counts blank lines, comment lines, and physical lines of source code in many programming languages. Given two versions of a code base, cloc can compute differences in source lines.

---

[3] http://radon.readthedocs.io/en/latest/intro.html
[4] https://sourceforge.net/projects/cloc/?source=typ_redirect

4.3 Maintainability Assessment Experiment

The goal of our experiment is to validate or refute the following hypothesis:

- *H1:* The selected maintainability metrics will show the difference between original and alternative model designs with respect to applied system level modifications;
- *H2:* The selected maintainability metrics will show the difference between original and alternative model designs with respect to applied component level modifications;

We compare two designs of the Heated Room model reported in [6]:

Set 1: Original design. We take the original system model (Case 0) and create new versions of this model applying system level modifications (Case 1) and component level modifications (Case 2) defined above (see Table 1). These modifications illustrate an adaptive maintenance of a DPRA model in order to reflect real-size system requirements. This set of three models is created following the original design ideas from [6] where system components are connected via PyCATSHOO message boxes, i.e., point-to-point.

Set 2: Alternative design. We create a set of semantically equivalent models of the Heated Room with alternative design (Case 0a - 2a). We promote the low coupling design principle by implementing well-known patterns of object-oriented design (i.e., Mediator, Observer [30]).

Maintainability Assessment and Analysis of Results. We use the metrics from Table 2 for both sets of models from Table 1 and analyse the results.

In the following sections, we explain the Heated Room test case, provide the details on this experiment and discuss the maitainability assessment results.

Table 2: Metrics for Maintainabiity Assessment of PyCATSHOO models

Metrics:	Tool support:
LOC (lines of code), RLOC (relative modifications)	cloc, radon
Cyclomatic Complexity (CC)	radon
Halstead (vocabulary, volume, effort, bugs, difficulty)	radon
Maintainability Index (MI)	radon

5 Case Study: Heated Room

"Heated Room" is the test case reported in [3]. This case is about a room which contains a heater device equipped with a thermostat. The latter switches the heater off when the room temperature reaches 20°C and switches it on when the room temperature falls below 15°C. The temperature is governed by a linear differential equation as: $dT/dt = \alpha T + \beta$ where α and β depend on the mode of heater. The heater is assumed to have a constant failure rate $\lambda = 0.01$ and repair rate $\mu = 0.1$.

5.1 Set 1: The Original PyCATSHOO Model of Heated Room

The original model of Heated Room is presented in [6]. It specifies the system components and their representing classes. The diagram in Fig. 1a shows the model from [6]. For the moment of this publication, the PyCATSHOO modeling tool does not have an explicit graphical modeling notation. We use the one that is adopted by EDF experts working with PyCATSHOO. The Heater class defines
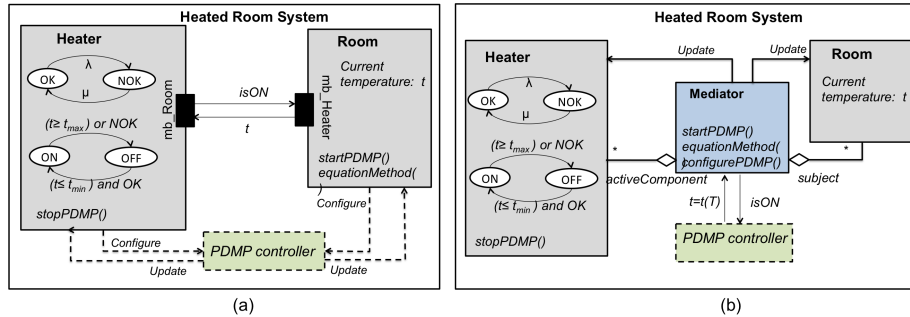

Fig. 1: Heated Room: a) Original design using message boxes. Point-to-point connection between system components; b) Alternative design using Design Patterns. Communication between components encapsulated by a Mediator

two automata: for functional and for dysfunctional behavior of the heater.

The Room class represents an observed subject: its temperature continuously evolves. The Heater and the Room communicate via message boxes: the Room component sends its current temperature to the heater, whereas the Heater sends its current state (ON or OFF) and its power value to the Room. In the original model design, the Room class contains the specification of a PDMP controller that implements the physics of the process - the evolution of the room temperature over time $T(t)$.

The PDMP controller is a part of the system. Other system components define the functioning of the PDMP controller in a distributed manner: *equationMethod()* for the room temperature is defined by the Room class, stop conditions *stopPDMP()* (e.g., when the boundary temperature value is reached) are defined by the Heater class.

In the initial model (Case 0), the Heated Room System class specifies the system with one heater and one room connected via corresponding message boxes (mb_Room and mb_Heater). The detailed specification of the model and the Python code listing can be found in in [6] and in our technical report [29].

System level modifications: In Case 1, we modify the structure and topology of the Heated Room system by instantiating four heaters (H0..H3) and connecting them explicitly to the room via message boxes. The Heater class does not change: the heater components independently heat the room following their initial control logic. Several heaters can be ON or OFF at the same time, based on the room temperature and their functioning state (OK or NOK). We modify the Room class and generalize the PDMP equation method in order to establish

the connection between the room and multiple heaters.

Component level modifications: In Case 2, we modify the control logic of the Heater component in order to support the standby redundancy mechanism: one heater (H0) will be declared as the main heater (the highest priority), whereas the other heaters will serve as its backups. Backup heaters are switching on only when all the other heaters with higher priority fail (NOK) and the temperature drops below T_{min}. If a heater with higher priority is repaired (OK), the backup heaters with lower priority switches off. In this case, only one heater at a time is heating the room.

We modify the Heater class: ON/OFF transition conditions; stop condition for the PDMP controller. We add new message boxes to communicate with other heaters. We modify the Heated Room System class where new heaters are explicitly connected to the room and to each other via message boxes.

5.2 Set 2: The Alternative Design of Heated Room

We integrate the low-coupling design principles early in modeling phase in order to anticipate component level and system level modifications. We use well-known design patterns from OO software development [30].

The *Mediator* design pattern encapsulates the interactions between system component and reduces direct dependencies between them. Integrating mediator early in the model improves system scalability.

The Mediator class maintains the lists of active components (heaters in our case) and subjects or passive components (i.e., rooms) as shown in Fig. 1b. It mediates the communication between the PDMP controller, the heater(s) and the room(s) replacing the point to point connections via message boxes. Note that an arbitrary number of rooms and heaters per room can be configured with this design.

The mediator contains the *configurePDMP()* method that allows to "assemble" the PDMP behavior from the parts defined by the active and passive components (i.e., *startPDMP(), stopPDMP(), equationMethod()*).

The Heater class is similar to the original design; the message boxes are removed. The Mediator object updates the heaters with the new value of the room temperature.

The Room class contains a current temperature variable that is updated by the Mediator component. Compared to the original design, the PDMP controller specification is moved from the Room class to the Mediator class. This allows to decouple the room and the heater.

In the initial model (Case 0a), the Heated Room System class specifies the system with one heater and one room attached to a mediator object. *configurePDMP()* method of the mediator terminates the system configuration.

5.3 Adaptive Maintenance of the Alternative Model: Case 1a, 2a

System level modifications: Case 1a is an alternative design of the Heated Room system with 4 independent heaters. The Heater, the Room and the Mediator classes are not changed. In the Heated Room System class new heaters are instantiated and attached to the mediator.

Component level modifications: Case 2a implements the standby redundancy algorithm for heaters. We modify the Heater class following the logic described in Case 2: ON/OFF transition conditions are changing; stop condition for the PDMP controller changes.

Compared to the original design, we do not connect heaters via message boxes, but implement the Observer design pattern.

The *Observer* design pattern allows for implementing specific behavior between a group of system components. It defines a one-to-many dependency between objects and when one system component changes state, all its dependents are notified and updated automatically. Due to space limitations, we do not provide the model details in this paper.

6 Maintainability Assessment: Results

We test the hypothesis from section 4.3 applying the metrics to the model sets representing the original and the alternative designs. In this paper, we report on application of the following metrics: Lines of Code (LOC); Relative modifications (RLOC) - a metric we propose, Cyclomatic Complexity (CC); Halstead's volume, difficulty, effort, estimated bugs (V, D, E, B); compound measure Maintainability index (MI). These metrics are computed statically from the code.

6.1 Size Metrics

LOC (Lines of Code) is a software metric used to measure the size of a computer program. It is recognised as a valid indicator of complexity and maintainability. We use LOC as a measure of the size of the PyCATSHOO model. The results of LOC measurement for two sets of PyCATSHOO models (Case 0-2, Case 0a-2a) are shown in Fig.2a. Stacked columns illustrate LOC per case; various colors correspond to various components. The following can be observed:
- The Mediator class in Set 2 doubles the size of the simplified model;
- For the Set 1, the size of all the classes is growing in response to system and component level modifications;
- For the Set 2, only the Heater and the Heated Room System classes are growing.
RLOC. We propose to measure relative modifications (RLOC) in response to system and component modifications. We use cloc tool to measure the difference between pairs of models (Case 0 and Case 1; Case 1 and Case 2) in both model sets. We calculate total modification as a sum of modified, added and removed lines in the "adapted" model. We define RLOC for a case as follows:

$$RLOC_{case} = \frac{LOC_{modif} + LOC_{add} + LOC_{rem}}{LOC_{case}}$$

Table 3: RLOC comparison: Original vs. Alternative design

Set 1	RLOC	Set 2	RLOC
$Case0 \longrightarrow Case1$	82,25%	$Case0a \longrightarrow Case1a$	18,57%
$Case1 \longrightarrow Case2$	108,69%	$Case1a \longrightarrow Case2a$	83,4%

This metric allows for more precise measurement of modifications as it takes into consideration not only the total change in model size. Removing or modifying the code (as well as model elements in concept model) are also considered. Table 3 summarises the RLOC measures. RLOC for the Set 1 shows that the model was modified on 82.25% to implement scalability (from 1 to 4 heaters) and further on 108.69% to implement standby redundancy of the heaters. While having bigger initial size (Fig. 2a), the model in the Set 2 was modified on 18.57% to scale for 4 heaters and on 83.4% to implement standby redundancy.
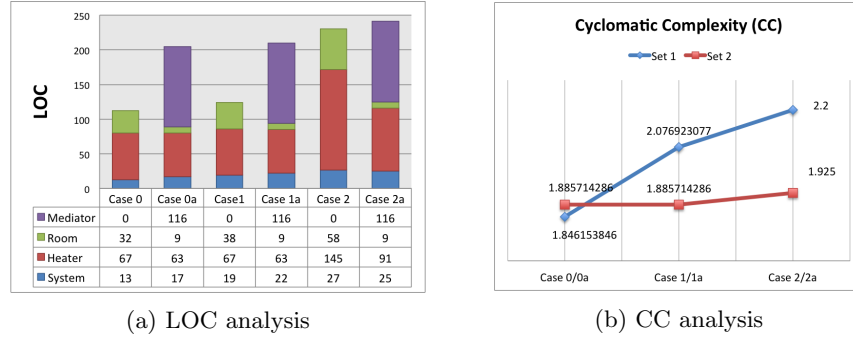

	Case 0	Case 0a	Case1	Case 1a	Case 2	Case 2a
■ Mediator	0	116	0	116	0	116
■ Room	32	9	38	9	58	9
■ Heater	67	63	67	63	145	91
■ System	13	17	19	22	27	25

(a) LOC analysis   (b) CC analysis

Fig. 2

6.2 Cyclomatic Complexity

McCabe [20] proposed Cyclomatic Complexity Measure to quantify complexity of a given software based on its flow-graph. A flow graph is based on decision-making constructs of a program. Fig.2b illustrates cyclomatic complexity measure for the model sets. Whereas the absolute CC values for all models indicate low complexity (below 10), we are interested in the CC evolution in response to adaptive model modifications. The graph in Fig.2b indicates the faster complexity growth for the Set 1. This corroborates with the previous measures.

6.3 Halstead lexical measures

Halstead metrics consider a program as a sequence of operators and their associated operands. For a given problem, let: η_1 = the number of distinct operators; η_2 = the number of distinct operands; N_1 = the total number of operators; N_2 = the total number of operands. From these numbers, several measures can be calculated. In this work, we measured volume (V), difficulty (D), efforts (E) and delivered bugs (B):

$$V = N \times \log_2 \eta, \quad D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}, \quad E = D \times V, \quad B = \frac{V}{3000}$$

Here D can be related to the difficulty of a PyCATSHOO model to write or to understand; B is an estimate for the number of errors. Fig.3 illustrates three metrics for two modeling sets and an evolution of these metrics in response to adaptive model modifications. The results show that an estimate number of errors (B) is higher for the Case 0a due to implementation of the Mediator;

however it seems to remain stable after adaptive modifications (Case 1a-2a) compared to the original design. Idem for the difficulty and effort measures.
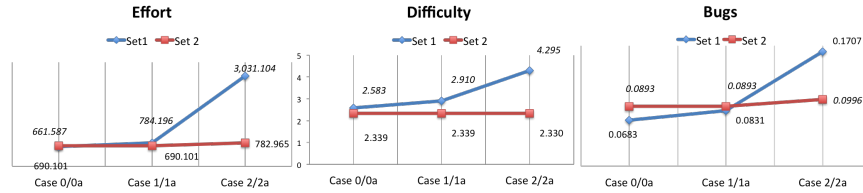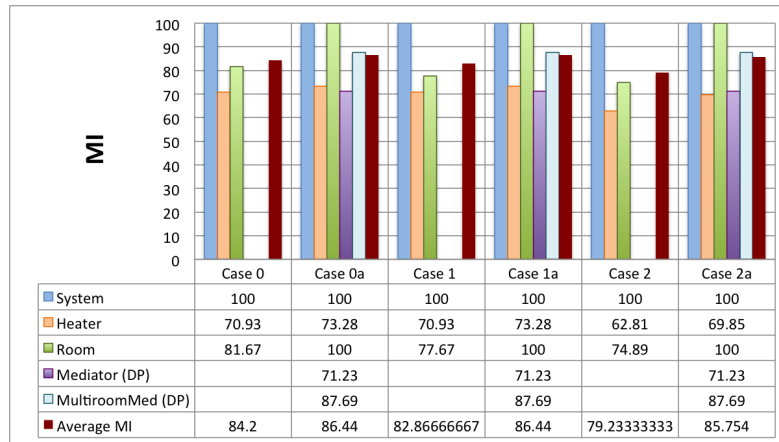

Fig. 3: Halstead analysis

Fig. 4: Maintainability Index analysis

6.4 Maintainability Index

Maintainability Index is a software metric which measures how maintainable (easy to support and change) the source code is. The maintainability index is calculated as a factored formula consisting of Lines Of Code (LOC), Cyclomatic Complexity (CC) and Halstead volume (V):

$$MI = 171 - 5.2 \times \ln(V) - 0.23 \times CC - 16.2 \times \ln(LOC)$$

Fig.4 illustrates MI measure for the model sets:

-Room class of the Set 2 and Heated Room System class of both sets are considered as 100% maintainable.

-MI of the Heater class decreases for both sets. This can be explained by its growing complexity and size.

-Average MI, though it remains very high for both sets, drops from 84.2 to 79.2 for the Set 1 and from 86.4 to 85.75 for the Set 2. More metrics and their discussion can be found in our technical report [29].

7 Conclusion

We proposed a pragmatic approach for maintainability assessment of DPRA models created with PyCATSHOO modeling tool. We claim that the adopted metrics can be used as early indicators for estimating maintainability of the PyCATSHOO models.

To validate our hypothesis, we created two model sets of the "Heated Room" system. Models in Set 1 follow the original design reported in [6] and based on point to point connection between system components. Set 2 promotes the low-coupling design principles. We used LOC, CC, Halstead and MI metrics on both sets of models.

The evaluation shows that the PyCATSHOO model based on design patterns requires more efforts at creation compared to the model that uses point-to-point connections. Nevertheless, complexity of this model grows slower when the number of components increases or when new types of dependencies are introduced. This indicates that the low coupling principle applied early in the modeling improves scalability and maintainability of the real-size DPRA model. The results corroborate with an engineering practice showing quantitatively the difference between designs even for a simple example of "Heated room" and thus validate our hypothesis. Based on this, we conclude that the selected metrics can be further considered for DPRA models assessment.

Despite the encouraging results, we consider them as preliminaries. As the next step, we plan to replicate our experiment and to assess the maintainability of the real PyCATSHOO projects. We also plan to extend the list of metrics integrating the metrics on Structural complexity and other OO-specific metrics.

References

1. Brito e Abreu, F.: The mood metrics set. In: proc. ECOOP. vol. 95, p. 267 (1995)
2. Aldemir, T.: A survey of dynamic methodologies for probabilistic safety assessment of nuclear power plants. Annals of Nuclear Energy 52, 113–124 (2013)
3. Bouissou, M., Chraibi, H., Chubarova, I.: Critical comparison of two user friendly tools to study piecewise deterministic markov processes (pdmp). ESREL (2013)
4. Cherfi, S.S.S., Akoka, J., Comyn-Wattiau, I.: Conceptual modeling quality-from eer to uml schemas evaluation. In: International Conference on Conceptual Modeling. pp. 414–428. Springer (2002)
5. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6), 476–493 (1994)
6. Chraibi, H.: Dynamic reliability modeling and assessment with pycatshoo: Application to a test case. PSAM (2013)
7. Chraibi, H.: Pycatshoo:toward a new platform dedicated to dynamic reliability assessments of hybrid systems. PSAM (2016)
8. Coyne, K., Siu, N.: Simulation-Based Analysis for Nuclear Power Plant Risk Assessment: Opportunities and Challenges. In: Proceeding of the ANS Embedded Conference on Risk Management for Complex Socio-Technical Systems (2013)
9. Genero, M., Manso, E., Visaggio, A., Canfora, G., Piattini, M.: Building measure-based prediction models for uml class diagram maintainability. Empirical Software Engineering 12(5), 517–549 (2007)

10. Genero, M., Miranda, D., Piattini, M.: Defining and validating metrics for uml statechart diagrams. Proceedings of QAOOSE 2002 (2002)
11. Genero, M., Piattini, M., Calero, C.: A survey of metrics for uml class diagrams. Journal of object technology 4(9), 59–92 (2005)
12. Gilb, T.: Designing maintainability in software engineering: a quantified approach. International Council on Systems Engineering (INCOSE) (2008)
13. Halstead, M.H.: Elements of software science, vol. 7. Elsevier New York (1977)
14. Harrison, R., Counsell, S.J., Nithi, R.V.: An evaluation of the mood set of object-oriented software metrics. IEEE Transactions on Software Engineering 24(6), 491–496 (1998)
15. Hoyle, D.: Iso 9000: quality systems handbook (2001)
16. IEEE: Standard glossary of software ùlengineering terminology. IEEE Software Engineering Standards Collection. IEEE pp. 610–12 (1990)
17. ISO/IEC: ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Tech. rep. (2010)
18. Li, W., Henry, S.: Object-oriented metrics that predict maintainability. Journal of systems and software 23(2), 111–122 (1993)
19. Marchesi, M.: Ooa metrics for the unified modeling language. In: Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on. pp. 67–73. IEEE (1998)
20. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
21. Meseguer, J., Sharykin, R.: Specification and analysis of distributed object-based stochastic hybrid systems. In: International Workshop on Hybrid Systems: Computation and Control. pp. 460–475. Springer (2006)
22. Michel, F., Ferber, J., Drogoul, A., et al.: Multi-agent systems and simulation: a survey from the agents community's perspective. Multi-Agent Systems: Simulation and Applications, Computational Analysis, Synthesis, and Design of Dynamic Systems pp. 3–52 (2009)
23. Moody, D.L.: Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data & Knowledge Engineering 55(3), 243–276 (2005)
24. Mylopoulos, J.: Conceptual modelling and telos. Conceptual Modelling, Databases, and CASE: an Integrated View of Information System Development, New York: John Wiley & Sons pp. 49–68 (1992)
25. Nelson, H.J., Poels, G., Genero, M., Piattini, M.: A conceptual modeling quality framework. Software Quality Journal 20(1), 201–228 (2012)
26. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Conference of Software Maintenance. pp. 337–344. IEEE (1992)
27. Riaz, M., Mendes, E., Tempero, E.: A systematic review of software maintainability prediction and metrics. In: 3rd International Symposium on Empirical Software Engineering and Measurement. pp. 367–377. IEEE (2009)
28. Rizvi, S., Khan, R.: Maintainability estimation model for object-oriented software in design phase (memood). arXiv preprint arXiv:1004.4447 (2010)
29. Rychkova, I., Boissier, F., Chraibi, H., Rychkov, V.: A pragmatic approach for measuring maintainability of dpra models. arXiv preprint arXiv:1706.02259 (2017)
30. Wolfgang, P.: Design patterns for object-oriented software development. Reading, Mass.: Addison-Wesley (1994)