Using Model Checking Approach for Grading the Semantics of UML Models

Hazim AlRawashdeh, Sufian Idris, and Abdullah Mohd Zin

Abstract—This study is concerned with the developing criteria for grading the semantics of UML models. This is achieved by going through literature and studying the current approaches for grading the semantics of UML diagrams. This paper concerns with the ability of grading the semantics of UML models in the logic of formal methods, where model checker or theorem provers can be run on these graphical diagrams. For this reason we integrate a transformation tool called Hugo/RT into our tool MUML that can help to map the model specifications and properties into Promela the C-like input programming language for Spin model checker. The last area is concerned with the integration of the graphic editors selection and the grading criteria designs with the functions of a CBA system. The proposed solution for this problem is to make provisions in the design and implementation of the CourseMaster CBA system as it has been used successfully in academia computer programming course work assessment. The result of this work is an enhanced environment for teaching and grading UML behaviour using the means of formal techniques. Evaluation results on diagram-based assessment do its expectations compared to some human evaluator of the students’ assignments, whereas circuit design and software design, show that the automation of diagrams assessment can be as useful and valuable as that for programs.

Keywords— UML semantics, Formal representation, Model Mapping, Spin, Hugo/RT

I. INTRODUCTION

MODEL checking (Gerard 2002) is a formal method for analyzing and verifying hardware and software systems. It is used generally for formal verification as it takes an automaton based model of a system and a temporal logic property as input then explores the entire state space of the system to determine whether the model violates the given property or not. Therefore it returned a counterexample to confirm the violation to the analyzer. The main idea behind it is that an abstraction of the system called a model is under inspection and not the whole system. Both desired and unwanted properties of the system can be logically formulated, whose type depends on the said tool. After that, both the model and its properties are presented in a suitable format to be used as input to a model checker, which tests the model and gives a feedback and reports conditions such as unreachable states and deadlocks, and conditions where the properties are violated. Later on the results obtained by the model checker can be used to improve and refine the model until it becomes free of errors.

Model checkers can be classified into two main types: symbolic and explicit, where symbolic model checkers are mostly used to model check hardware. The states of the system are encoded. Explicit state model checkers, on the other hand, is used typically used to model check software systems. It examines the states that are stored explicitly, each in turn.

There are a lot of work efforts that describe how to convert a UML model into a Promela specification. Promela is the input language of the SPIN model checker (Gerard 2002). The SPIN software verification tool, designed in mid-eighties by Gerard (2002) in AT&T Labs, is based on an interleaving model of concurrency where only one component of the system’s state is allowed to change at a time. The interleaving semantics supports a reduction algorithm based upon exploiting symmetries in the order of execution known as partial order reduction (Miller et al. 2006). This makes SPIN suitable for concurrent processes with uncoupled events. For most reactive systems where events are tightly coupled in distinct processes, SPIN’s partial order reduction technique cannot be put to good use. On the other hand, the symbolic model checker SMV, is used for verifying the properties, as it is one of the most popular model checkers that supports CTL and it has been used for model checking digital circuits security protocols embedded systems and web applications (Bohlin et al. 2009).

II. RELATED WORK

In testing perspective, the most used model checkers are the explicit state model checker SPIN (Simple Promela Interpreter) (Gerard 2003), the Symbolic Analysis Laboratory (SAL) (Leonardo et al. 2004), which can support both bounded and symbolic model checking, the symbolic model checker SMV as well as its derived NuSMV (Alessandro et al. 1999), which supports both bounded and symbolic model checking. There are many model checkers such as ProbVerus (Garmhausen et al. 1999), which supports model checking, of DTMCs using a subset of PCTL, and COSPAN (Hardin et al. 1996); but not all of them are used for testing.

A. The SPIN Model Checker

SPIN model checker is a tool whose models are written in Promela language (Process or Protocol Meta Language) with asynchronous communicated process. SPIN was designed
basically for protocol verification, and soon after and perhaps because of the model’s nature it uses and the appearance of various programming languages, it has been used in a big variety of problem domains. It isn’t only applied on protocols, but it has been applied in embedded systems modeling, verification of requirements and design as well. A study of prediction of inherited and genetic mutations the authors Zubin and Steve (2005) have used SPIN for checking genetic mutation and modeling signaling pathways. They put their hands on the similarity of undesirable mutations in gene sections and errors in computer programs. So the individual mutations are modeled as processes, and express as a property that gives a certain kind of gene sequence as input.

B. The Symbolic Model Verifier (SMV)

Symbolic model verifier: this is developed by McMillan and it was some work of his PhD thesis is called SMV (McMillan 1993). It is in fact based on a programming language that describes the hierarchical finite-state concurrent systems. The programs written in this language can be interpreted by specifications written in temporal logic. SMV can extract the presentation of transition system as an ordered binary decision diagram (OBDD) from a certain program in the SMV language and uses one search algorithm based on OBDD to decide whether the given system satisfies its specification or not. In case of non satisfaction for some specification, SMV will create a counter example for the execution trace that shows why the specification is false.

The SMV model checker has been widely distributed, and it has done much verification on several systems. These multiple verification processes provide compelling indication that SMV can be useful for debugging actual industrial designs. Now a days there are some widely used releases of SMV such as Cadence SMV; released by Cadence Berkeley Labs and an open source version, called NuSMV (Clarke 2007), which is released by IRST in Trento, Italy. The verification of cache coherence protocol IOMG 2 Futurebus + standard reflect the strength of the SMV model checker.

C. The Approximate Probabilistic Model Checker (APMC)

APMC uses a randomized algorithm to approximate the probability that a temporal formula is true, and this is done using execution sampling paths of the system (Herault et al. 2004). Thomas et al. (2006) showed that APMC uses a distributed computation model to distribute path generation and formula verification on workstations cluster. APMC is described in its website (Herault et al. 2006) as an implementation of an algorithm to adapt the problem of the LTL quantitative verification. There are two versions available at the same time: APMC, APMC 3.0 and CA. The only difference being that the latter are adjusted by hand to work on architectural CELL. Input language is the same as Probabilistic Symbolic Model Checker (PRISM) and APMC control model of discrete or continuous time Markov chains. APMC can be used for calculating the estimated value of the probability of the temporal property of a probabilistic system. In fact APMC works with the generation of random number paths in the probabilistic space of the system, and a random variable which evaluates the nature of the calculated probability.

Finally, there are so many model checkers, where some of them are rarely used, and it is out of the scope of this study to handle all of them, but some of them can be listed as examples. For instance, Markov Reward Model Checker (MRMC) as described by Katoen et al. (2007) is a model checker for discrete-time and continuous-time Markov reward models. It allows for the automated verification of properties concerning long-run and instantaneous rewards as well as cumulative rewards (visiting legal states before) under a time and an accumulated reward constraint.

D. FORMAL REPRESENTATION OF UML MODELS

1. Hugo and Hugo/RT

A research of Model Checking and Code Generation for UML State Machines and Collaborations uses HUGO project (Knapp et al. 2002). The project consists of some tools used to apply the model checking process to UML state and collaboration diagrams. Then the tool is used later to check and verify whether the interaction represented in the collaboration diagram can be performed using state machines and it uses XMI to automate the mapping process of UML to SPIN. It is another translator of UML 2.0 interactions into automata, and the translation process integrates UML model with model checking tools (Knapp and Merz 2002). Knapp and Zhang (2006) showed that a system is represented by message exchanging the state machines of the UML model with the automaton representation of UML interaction to observe the message traces to be translated into the input language of SPIN or UPPAL, to check the satisfiability of the given model.

It is out of this study scope to cover all the automatic verification techniques, since a lot of work on this was done. For instance Rebeca is an actor-based language produced by Sirjani et al. (2004) for modeling and verification of reactive systems. The key features of Rebeca are: using actor-based concepts for the specification of reactive systems and their communications, providing a formal semantics for the model, providing a tool for model checking Rebeca code and using abstraction techniques to reduce the state space in model checking. On the other hand, Guennec and Dion (2006) have also a study about UML state diagrams encoded through hierarchical automata (HA) for generating SPIN specifications, where the mapping processes have been carried out manually and later on the automated work has achieved using XMI.

2. STAIRS Approach

Øystein et al. (2003) have presented an approach to the compositional development of UML interactions for the specification of mandatory and potential behavior of the model called STAIRS (Steps to Analyze Interactions with Refinement Semantics) that have been designed to facilitate the use of interactions in requirement capturing specification test. STAIRS basically do formal methods of refinement. This
approach is described in section 5.5.2.

3. AREDO Framework

It’s a framework for representing formal semantics of a subset of the Unified Modeling Language (UML) notation in higher-order logic; more specifically semantics of UML sequence diagrams is encoded into the Prototype Verification System (PVS). PVS is a tool that integrates UML and PVS system so the UML modeling constructs including Class, Sequence and Statechart Diagrams can be mapped into the specification language (PVS) for verification by its type-checkers, Theorem-Provers, and model-checkers (Aredo 2004).

4. UML2ALLOY Approach

UML2Alloy is a filter tool for formatting UML Class Diagrams enriched with OCL formulas as Alloy specifications. Anastasakis et al. (2007) state that the current version of UML2Alloy performs the translation creating a text file; the designer, which knows UML and OCL may not have any notion about Alloy language syntax, only needs to use the Alloy Analyzer to open the text file and perform the analysis.

A work was done for building a some methods in a way that a design model must be transformed to a language suitable for analysis (Shah et al. 2009), and after conducting analysis the result must be transformed back to the design space. Alloy (Seyyed et al. 2009) is used also to analyze the model in order to identify inconsistencies that might occur in the design. For instance, if there is some inconsistency among several parts of the design, Alloy can produce a counter-example that helps to show the source of that inconsistency. This can help in recognizing such inconsistencies, which can help to improve the design at earlier stages of software development process.

5. The PUML Approach

This approach carried out by the precise UML (pUML) group and documented in the pUML web site (Andy and Stuart 1999). It is an international group of researchers who share the goal of developing UML as a precise (formal) modelling language. Andy and Stuart (1999) have a report on their work that aims to strengthen the existing meta-model semantics of UML. In their work they described a formalization strategy (developed through the experiences of the group) that is being used to precisely describe the semantics of UML. The aim of the study was to give a precise denotational semantics to the core elements of UML.

The work in this study is consistent with their homomorphic mapping approach because it preserves UML’s class structure when translating from UML to SPIN. Our work is Similar in that this work also considers translating UML models into SPIN and not to SMV. Hence we provide specific rules on how to map UML classes and attributes to SPIN’s modules and variables. Table 1 gives the description of some of the UML formalization methods.

III. METHODS

Going through literature review can make a good insight about approaches that are used for verifying the formal mapping of UML models. Current verification tools have been reviewed and some of them have been installed and tested, such as TABU. In this study Hugo/RT is used to transform UML models for model checking and code generation, as UML models can contain active classes either with state machines or collaborations, then these active classes could be mapped into the system language of Spin model checker for performing formal verification (Knapp and Merz 2002).

To demine which model checker to be used for measuring and verifying the semantics of UML diagrams a review of literature was done in this stage. For verification purpose, the most used model checkers are SPIN model checker (Gerard 2003), SAL model checker which can support both bounded and symbolic model checking (Leonardo et al. 2004), and SMV model checker as well as its derived NuSMV (Alessandro et al. 1999).

Based on the study of the available formalization approaches and model checking tools, a list of possible tools that can be used for measuring and verifying UML models are selected. Some of these tools are installed and tested and based on that the reasonable tools are then selected. In this study Hugo/RT is selected for achieving the formalization process for the UML models, and Spin model checker is selected to measure and verify the generated formal model.

A framework for measuring and verifying the semantics of UML models is developed by going through the next processes:

- Identify steps required for verifying semantics of UML models and this is done by mapping UML model elements into the formal notation using the verification tools that are chosen, and then identify a technique for measuring the semantic correctness of UML models.

<table>
<thead>
<tr>
<th>Produced by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aredo</td>
<td>Integration tool for UML and PVS</td>
</tr>
<tr>
<td>Brained et al.</td>
<td>Detection rules for UML designs inconsistencies</td>
</tr>
<tr>
<td>Dong</td>
<td>Integrating formal languages into semantic Web environment</td>
</tr>
<tr>
<td>Jin</td>
<td>Ontology and tool adapters provide interoperability of software reengineering tools</td>
</tr>
<tr>
<td>Kalfoglou</td>
<td>Ontology to identify conceptual errors in software specifications</td>
</tr>
<tr>
<td>Kitamura &amp; Mizoguchi</td>
<td>Ontological organization of functional design knowledge</td>
</tr>
<tr>
<td>Liu</td>
<td>Rule-based inconsistency classification</td>
</tr>
<tr>
<td>Perini</td>
<td>Integrates agent-oriented modeling tool with software verification tool</td>
</tr>
<tr>
<td>Silva &amp; Lucena</td>
<td>Combines concepts of agents, objects, and UML into a multi-agent modeling language</td>
</tr>
<tr>
<td>Vergara et al.</td>
<td>Agent-based requirements refinement model including a domain ontology; detect, diagnose, resolve inconsistencies in software requirements</td>
</tr>
</tbody>
</table>

http://dx.doi.org/10.15242/IIE.E0114567
model checker output result, while in this study we use the TRAIL file of SPIN model checker that contains the result of semantics check.

- Developing and building the framework; the framework for handling this method is to use Hugo/RT formal mapping technique that convert student diagram into temporal logic format to capture the properties and convert it again into Promela language file. Promela which contains the elements and properties of the original UML model is used as an input for the Spin model checker, whereas a result of Spin execution is stored in a Trail file.

- Test and evaluate the proposed framework; the test of the framework is done by doing the conversion process manually by experts (it takes more time, especially if the state space of the model is huge). The evaluation is made by doing the conversion for some sample models, once by using the tools and the second time is manually by experts in order to compare the outcomes.

All these processes mentioned above are used for the evaluation process to guarantee the verification of UML model semantics. Figure 1 shows an example of an actual question. The question should contain obvious names for active actors or classes which would make it easy for student to pick them from the text. Furthermore, a description for the system behavior should be described in detail. The example in Figure 1 is written in a simple way that makes it easy for students to identify the elements of the model. For example, it is clear that the answer should have at least one ATM class and one Customer class. It is also clear that the behavior of the system should include some operations based on some conditions, such as (Enter code, Abort operation, Enter amount, Print receipt and Withdrawal).

A. Semantics Checking

There are so many approaches for checking the semantics of UML model (Harel and Politi 1997). Our approach for the semantics check is slightly similar to the one proposed by Beato et al. (2004) which is called TABU and also to STAIRS approach (Seehusen et al. 2009). The similarity is that the semantics of the UML model is checked using model checking technique. In TABU, the framework for the formal representations of UML model behavior uses class diagrams, activity diagrams and state diagrams as the input. They are mapped into a formal format which is suitable for the SMV model checker. Our approach differs from TABU by using sequence diagrams instead of activity diagrams for representing the model behavior and it does not only verify the semantics but the syntax and relevancy as well.

B. Formal Model Verification

Formal UML models are verified using model checking techniques. Many model checkers can be used for semantic verification. The Spin model checker is chosen for this study. The reason for this choice is that it specifically verifies software unlike other model checkers which also verifies hardware. For example, the target of SMV model checker is the verification of hardware circuits, while the target of UPPAAL model checker is the verification of real-time systems.

Secondly, Spin is open source and multi-platform, distributed for free as a research tool. It has a good documentation with a large user base in both academia and industry. It has its also owns annual workshops held since 1995 (Daniel and Guido 2008). Spin model checker has a user friendly graphical user interface that allows the user to write directly the system specifications using Promela the C like language. Promela is the input language for SPIN where it can be edited inside the tool or imported from the file location. Figure 1 shows an example of Promela code written in jSpin which is a version of Spin model checker. The interface of jSpin is divided into three parts, one for showing and editing the code, and one part for the command line option and the third part for the output. The output is therefore saved in a TRAIL file which is empty if there are no execution errors. Figure 2 shows the interface of jSPIN model checker.

Since the input for Spin is Promela therefore, UML model needs to be translated to Promela. Hugo/RT is used in this work for mapping the UML model to Promela language. It is used to transform UML models for model checking, theorem proving, and code generation. Since UML models can contain active classes either with state machines, collaborations or with both, then these active classes can be mapped into the system language of Spin model checker and some other Theorem Provers (Knapp and Merz 2002).
C. Semantics checking and grading

This section describes the process for the semantics check. The input for this particular part of the grading tools verifies some LTL properties. Instructors can insert these properties directly along with question, or by inserting them in the specific panel in jSpin. The grading tool stores them in a file called property.txt.

For the example described in Figure 1, some properties where chosen by the focus group members to verify the semantics of the ATM machine students model. Table 2 describes these properties and the correspondent specifications.

<table>
<thead>
<tr>
<th>No</th>
<th>Property</th>
<th>Specification</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The ATM cannot allow the user to request an operation if either the card or the PIN is not valid</td>
<td>(x \rightarrow \neg y)</td>
<td>(x = (\text{start} \land \neg \text{cardOK} \lor \neg \text{PINok}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(y = (\text{proc1 user} \land \text{receive} \land \text{msg waitAccount}))</td>
</tr>
<tr>
<td>2</td>
<td>ATM must first debit the amount in the bank, and then give the money to the user. In other words, the user does not receive pickCash until the bank receives debit</td>
<td>(\neg x \land y)</td>
<td>(x = (\text{proc1 user} \land \text{receive} \land \text{msg pickCash}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(y = (\text{proc1 bank} \land \text{receive} \land \text{msg debit}))</td>
</tr>
<tr>
<td>3</td>
<td>If the ATM receives insufficient funds, it should allow the user to choose other operation before finishing the session</td>
<td>(x \rightarrow (\neg y \land w))</td>
<td>(x = (\text{proc1 user} \land \text{receive} \land \text{msg insufficientFunds}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(y = (\text{end})), and (w = (\text{proc1 atm} \land \text{send} \land \text{msg waitOperation}))</td>
</tr>
<tr>
<td>4</td>
<td>Ensure the correct end of the session between the ATM and the user. It says that, after the user receives ejectCard, the ATM cannot send anything to the user</td>
<td>(x \rightarrow \neg y)</td>
<td>(x = (\text{proc1 user} \land \text{receive} \land \text{msg ejectCard}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(y = (\text{proc1 atm} \land \text{send} \land \text{proc2 user}))</td>
</tr>
</tbody>
</table>

Fig. 2 An Example of jSpin Model Checker GUI
There are so many semantics elements in the model, but elements that are normally checked include:

- The relations between classes
- The states of each class object
- The activities of the model
- The sequences of model behavior
- The LTL properties given by instructors, if any

The task of developing the framework for grading the semantics of UML models, besides checking their relevancy and syntax is to check whether the answer to the given question submitted by a student is semantically correct. To do this, the submitted answer needs to be checked against the properties given by the instructor. The semantics check is to be performed based on the focus group recommendations by the following algorithm in Figure 3:

Fig. 3 The Semantics Check Algorithm

The process of converting a zargo file into Promela is to be done by using a tool called Hugo/RT. Hugo/RT is a message exchanging system for UML state machines. It generates an automaton representing a UML interaction for observing message traces. These traces are then translated into the input language of Spin model checker, which is then called upon to check for satisfiability. Hugo/RT translates the model into a set of SPIN processes and calls SPIN for finding acceptance cycles. The interactions of UML model are represented in the input language of HUGO/RT. UTE, a textual UML description language, is used to represent all UML features which are extracted from the model.

The following command is an example of converting a zargo file (file.zargo) into Promela (file.pml) in the DOS operating system environment:

```
java –jar hugort.jar –outputpromela -promela= file.pml file.zargo
```

To complete the semantics check the Spin model checker is used. Spin is based on an interleaving model of concurrency where only one component of the system’s state is allowed to change at a time. The interleaving semantics supports a reduction algorithm based upon exploiting symmetries in the order of execution known as partial order reduction. This makes SPIN suitable for concurrent processes with uncoupled events such as network protocols. Spin model checker can be run to check the model by calling the following command:

```
Spin.exe –a –v file.pml
```

The summary of the output from the Spin model checker for the four properties given by the instructor in Figure 1 is shown below in Table III.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Message produced</th>
<th>Full Mark</th>
<th>Mark Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ok</td>
<td>0 errors, 2 warnings</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>Produced TRAIL</td>
<td>5 errors</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>ok</td>
<td>No errors</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Produced TRAIL</td>
<td>7 errors, 8 warnings</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>

From the result in that table, it is clear that a total of 12 errors and 10 warnings have occurred. Based on semantics grading algorithm and the mark distribution file for the ATM example, the semantics mark can be calculated as follows. Since the allocated marks for semantic are 60% of the total marks (40% for the relevancy and syntax).

As a result, Spin model checker could execute Promela with an empty TRAIL file. That means the total mark for semantics will be 100%. On the other hand SPIN will generate a TRAIL file or a Counter example, which means that error(s) in some properties are found. In this case the allocated semantics mark will be reduced by (100 / no. of properties) for each incorrect property, and can be calculated as follows for the given question in Figure 1:

\[
\text{semMark} = 100 - \left(\frac{\text{no. of true props}}{\text{total no. of props}} \times 100\right)
\]

\[
\text{semMark} = 100 - \left(\frac{2}{4} \times 100\right)
\]

\[
\text{semMark} = 50
\]

Where semMark stands for semantics mark, and props stands for properties.

D. Overall Mark Calculation

Sections 5.3 to 5.4 describe in detail the calculation of marks for relevancy, syntax and semantics of UML models. The calculation of the overall mark is obtained by student based on the marks distribution given in the mark file. For example, suppose that the mark file contains the following information:

1. Relevancy: 20%
2. Syntax: 20%
3. Semantics: 60%

The overall mark will be calculated as shown in Table 4 based on the weightage provided by the instructor in the mark file, where he can determine the weightage for each category
based on its importance for the given question.

<table>
<thead>
<tr>
<th>Check type</th>
<th>Weightage</th>
<th>Marks obtained</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevancy</td>
<td>20%</td>
<td>83.3</td>
<td>16.6</td>
</tr>
<tr>
<td>Syntax</td>
<td>30%</td>
<td>63.6</td>
<td>12.6</td>
</tr>
<tr>
<td>Semantics</td>
<td>50%</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>Overall</td>
<td>100%</td>
<td>59.2</td>
<td></td>
</tr>
</tbody>
</table>

### IV. CONCLUSION

The development of a framework for grading the semantics of UML models helps instructors to do the challenging part of grading UML diagrams i.e. the behavior of the system that needs a big effort from them if done manually, as well as the relevancy and syntax grading. The paper also showed how to benefit from the output of certain model checkers to determine the mark of this critical part of the question, where applying some properties to the model can’t be checked manually.

The approach in this study make use of the above described software tools and combine all of them i.e. ArgoUML, Hugo/RT, Spin model checker under one interface for grading the relevancy, syntax and semantics of UML models submitted by UML course students. The paper briefly illustrated the framework for grading the relevancy of a given solution to assure whether it is the desired question or not, and to obtain mark for this particular part of the overall grading tool. The paper also described the framework for grading the semantics of the given answer, by applying some rules recommended by the focus group. And based on these rules the mark for this part is obtained. This paper discusses how to grade the semantic for the given answer, where it explains how the model is mapped from UML diagrammatic representation into a formal representation. The formal representation of the model answer allowed us to use the model checkers to be run on the transformed model. The result of the model checker was used to determine the mark that should be obtained for the semantics.

Finally, the paper described how to obtain the weightage for each part of the grading aspects and how to obtain the overall mark for the given model answer.

### REFERENCES


[16] http://dx.doi.org/10.1007/978-3-540-24622-0_8

[17] http://dx.doi.org/10.1007/978-3-540-45739-9_23

http://dx.doi.org/10.15242/IIIE.E0114567

128
http://dx.doi.org/10.1145/1132960.1132962


