Formal modeling and proving of Campus Management System: Event-B perspective

Nadeem Akhtar, Malik M. Saad Missen, and Rida Zahra Hashmi

***Abstract* – Campus Management System (CMS) provides the management and information processing services that are critical for the efficient working of the university. A CMS is a complex system formed by the integration of a number of interacting sub-systems working together. Errors at any level in CMS can cause huge loss, therefore it is important to ensure correctness of the system. A CMS has been formally specified, modeled, and validated for the Baghdad-ul-Jadeed campus of The Islamia University of Bahawalpur, Pakistan. It provides a formally validated correct platform for automation and management of all the aspects of student admission, examination, student attendance, results and faculty attendance. This CMS is based on formal modeling and formal proofs to ensure correctness, with model-based methods with underlying mathematical concepts of set theory and first-order predicate calculus. A novel abstraction and refinement based formal method Event-B is used. It has an exhaustive industrial development platform RODIN which provides exhaustive evaluation and implementations. The proposed CMS model is centered on the fundamental principles of abstraction and refinement.**

***Index Terms* – Campus Management System (CMS); Event-B; Formal modeling; Formal validation; Theorem Proving; RODIN.**

I. INTRODUCTION

Education plays one of the most important roles in the development of a nation. Universities provide higher education as well as scientific progress to masses. They impart the education and technical skills required for the jobs in the industry. They are fundamental in improving the economic situation, as a result improving the quality of life of people. A CMS has been proposed for the Department of Computer Science & IT, The Islamia University of Bahawalpur, Pakistan. The department has about ten thousand students enrolled in a number of programs i.e. MCS (Master in Computer Science), BS (Computer Science), BS (Information Technology), MSCS (Master of Science in Computer Science), and PhD (Computer Science).

The analysis, design, formal modeling and proving of a correct Campus Management System (CMS). The correctness of the system is ensured by using formal modeling and specifications centered on abstractions and refinement. The CMS is built by first building an abstract model of the system, and then this abstract model is refined into a more detailed model (i.e. another abstraction level). There are a number of refinement processes starting from a very abstract model of the CMS, periodically making refinements in this abstract model making it a more detailed one, and at the end after a number of refinement layers resulting into a very detailed model of the CMS. This detailed model is then implemented by using a high-level implementation language.

The major objectives are:

1) Formal requirement specifications of a CMS in the form of an abstract model.
2) Formal design specifications of the CMS centered on multiple refinements of the abstract model i.e. multiple refinement passes applied on multiple abstract models. Moving from abstract models towards more concrete models.
3) Proof rules based verification of the correctness properties of the proposed CMS models at each and every refinement level i.e. from abstract levels to concrete levels.
4) UML-B based formal diagrams for the exhaustive investigation of states and transitions.

II. STATE OF THE ART

Formal modeling and formal proofs are important to ensure the correctness of critical systems. Errors in critical systems can cause human life loss. Event-B [1] [2] is a novel formal proof-by-construction method, centered on the fundamental software engineering principles of refinement and abstraction. It is a proof-based formal method ideally suited for safety critical systems. In the last few years there are some very interesting work carried out in formal-proofs by using Event-B with its toolset rodin. [3] has argued that software safety criteria plays a very important role in verifying software safety, therefore software safety criteria has been formalized and verified by using Event-B. [4] have proposed an approach that uses structured natural language conformant to the formalism of the Semantics of Business Vocabulary and Business Rules (SBVR) standard for the implementation of automated transformation from requirements document to a formal specification in Event-B. [5] have proposed a model based testing approach for reactive systems where both the test inputs and expected results are generated from "restricted" Event-B models. This work has proved that it is possible to automatically build the restricted Event-B models from the knowledge base of the system under test. These restricted models reduce the state space of the original Event-B models while preserving the possible testing paths. [6] have proposed an approach for the specification and verification of flexible workflow applications of cloud services. A tool is proposed for automated development of sequence diagrams and then transformation into Event-B model for formal verification.

Nadeem Akhtar and Malik M. Saad Missen are with Department of Computer Science & IT, The Islamia University of Bahawalpur. Rida Zahra Hashmi is with Department of Software Engineering, Bahria University Islamabad. Email: nadeem.akhtar@iub.edu.pk, saad.missen@gmail.com, rida.zahra@bahria.edu.pk. Manuscript received on Sep 11, 2017 revised on Nov 16, 2017 and accepted on Dec 22, 2017.

 ISSN – 1999-4974

[7] summarized experiences of teaching formal methods course of Models of Software Systems (MSS) to master programs at Carnegie Mellon University (CMU) USA and Innopolis University (INNO) Russia. The benefits of teaching Event-B language and its underlying software development methodology to students are highlighted. This Event-B course goal is to create leaders in the field of Software Engineering. [8] works to ensure correct design of the web service composition. simple or complex web services can be composed to build more complex web services. Thus web services composition creates complexity. The problem of the correct design of a web service compositions in case of failures is important. This work presents a novel correct-by-construction formal approach based on refinement using the Event-B method.

III. FORMAL MODELING AND PROVING

A formal model ensures correctness by performing a rigorous analysis of the system. It improves the system quality and allows the formal specifications to be reused in the implementation. In a development approach based on formal modeling, major emphasis is put on the specification phase (i.e. the specification in relation to the requirements is verified, and by using formal theorem proving it is ensured that the specification is consistent with the requirements); minor emphasis is also put on the test phase verification of the implementation. In the construction of a complex discrete system the most important activity in terms of time and money, is the formal proving that the implementation is functionally correct i.e. final behavior is consistent with the requirements.

Program testing used as a validation process is far from being a complete rigorous process, because of the impossibility of achieving a total cover of all executing cases. There is incompleteness as a consequence of the lack of oracles. Oracles generate the expected result of a future testing session, beforehand and independently of the tested objects. Testing does not involve any kind of rigorous exhaustive investigation during the requirement specification and design phase. Testing always only gives the operational view of the system under construction.

The CMS system should eventually be tested, but testing is the routine evaluation of the implementation process. The important part is the correct-by-construction approach based on formal models. Therefore the formal specification and proving is the fundamental phase of the construction; and in this phase most of the validation and proving is done well before the implementation of the final system [1].

IV. EVENT-B

Event-B [1] [2] is a formal method for constructing and validating a precise, accurate, rigorous model of a system. This model is gradually refined into a more detailed model with the help of a number of refinement levels.

The Event-B notation is based on set theory; it formally models the system centered around abstract machines; it uses the principle of refinement to represent the system at different abstraction levels; it formally verifies consistency between these refinement levels by using mathematical proofs. Event-B can be used for the construction of complex discrete systems. It models system in a discrete fashion. The behavior of a CMS model is continuous; the systems operate most of the time in a discrete fashion. The CMS behavior can be abstracted by a succession of states. In reactive systems, transitions are occurring concurrently and rapidly, resulting into large number of concurrent changes occurring at a very high frequency. Despite a high number and frequency of changes, such systems are intrinsically discrete. They are also called transition systems.

Event-B involves modeling and formal reasoning by constructing a mathematical model which will be analyzed by mathematical proofs. The initial model of the CMS specifies the properties that the system must fulfill. Modeling is accompanied by reasoning. Model of a program also contains proofs that are related to the properties of the program. The challenge is to incorporate formal models in the analysis and design of CMS and validate the correctness properties.

CMS is complex and it is made up of many parts interacting components in a dynamic environment that continuously evolves. It requires a high degree of correctness. Complex models are built by the method of step-wise refinement i.e. a model is built by successive refinements of an original simple machine carefully transforming it into more concrete machine. An Event-B model consists of contexts and machines.

- A Context specifies the static parts of the model. It contains sets, constants, axioms, and theorems.
- A Machine specifies the dynamic components of the model. It has a state, defined variables. A variable is specified by mathematical objects i.e. sets, binary relations, functions, numbers etc. A variable is constrained by an invariant. Invariants are specified to hold whenever variable values change.

In order to design the CMS a discrete model is made of the real system; this model is designed at a certain level of abstraction; then periodically this discrete model is detailed with the help of number of refinements. This discrete dynamic model constitutes a kind of state transition machine. The model consists of a number of states, and transitions that are triggered under certain circumstances. These transitions are called "events", and are an integral part of Event-B. Each event is composed of a guard and an action. "The guard is a predicate built on the variables and state constants. It specifies the condition under which the event may occur. An event may be executed only when its guard holds. The action, signifies the way in which state variables evolve when the event occurs. An event has parameters which can be used to model array of events or communication channel in the composition of machines. Events have guards which are conditions that must be true when an event should execute. When the guards of more than one events are true at the same time, then one of the events is executed, this choice is made in a non-deterministic fashion" [1]. Two events cannot occur simultaneously. The execution is as follows: "When no event

 ISSN – 1999-4974

guards are true, then the model execution stops; the model has deadlocked. When some event guards are true, then one of the corresponding events occurs and the state is modified accordingly; subsequently, the guards are checked again, and so on. When only one guard is true at all times, the model is said to be deterministic. It's not mandatory for a model to eventually finish. Most of the systems never deadlock; they run forever" [1].

D. *Formal Reasoning*

There are two kinds of discrete model properties.

1. Invariant property are proven about models and ultimately about real systems. "An invariant is a condition on the state variables that must hold permanently. The invariant must hold under the guard of each event" [1].
2. There are reasoning by conditions called modalities which do not hold permanently. A special form of modality is reach-ability.

F. *Refinement*

Refinement builds a model gradually by making it more and more precise. An initial model of the CMS representing all components is not built on the sudden once and for all. It is built gradually in the form of a number of refinements and abstractions. A sequence of embedded models is constructed, each embedded model is a refinement of the previous one.

G. *Decomposition*

The process of decomposition reduces complexity. A single model is divided into a number of component models in a systematic way.

H. *Rodin*

Rodin [2] [9] platform is an Eclipse-based Integrated Development Environment (IDE) for Event-B [1] that provides construct for the implementation of refinement and mathematical proofs. It is the implementation platform of Event-B; it is founded on the mathematical concepts of set theory, predicate logic, relations, and functions. It integrates

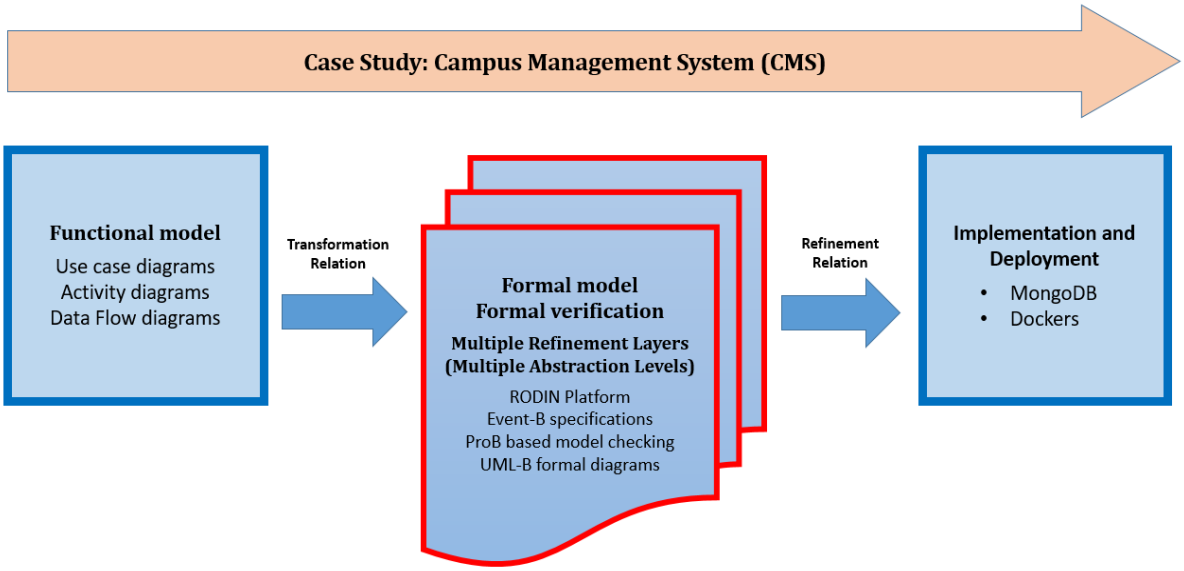

Fig. 1 Block diagram of the proposed CMS approach

E. *Managing the complexity of closed models*

A model built in Event-B specifies the controller of the system as well as the environment within which the controller works. A closed model specifies the actions and reactions taking place in the controller and environment. The model of the controller is inserted within the model of an environment. The transitions are of two types: those related with the environment and those related with the controller. The communication between these two entities are also modeled. The number of variables describing the state of such a system is also large. This complexity is managed by using three fundamental concepts of refinement, decomposition, and generic instantiation. These concepts are linked together. A CMS model is refined to later decompose it, and, it is decomposed further to refine it. Finally, as a result of a number of refinements and decompositions a detailed model of the system is developed which can then be instantiated.

modeling as well as exhaustive proving. It has made a large contribution in making theorem proving a practical tool for software verification and validation. It implements techniques used in programming, formal modeling to formal verifications. Instead of compilation, Rodin is centered on proof obligation generation and automatically discharging trivial proof obligations. It analyzes, validates and reasons about CMS models. ProB [10] is a model checker for Event-B language that can be integrated in Event-B.

V. THE PROPOSED SYSTEM: CAMPUS MANAGEMENT SYSTEM

A CMS is formally modeled and validated. This system manages the teachers, students, programs, and courses offered in the Department of Computer Science & IT, The Islamia University of Bahawalpur. This proposed CMS has a mathematical foundation, and is proven by the automated

 ISSN – 1999-4974

proof obligations of RODIN. This modeling and proving using formal models and correct-by-construction ensures correctness. The software engineering principles of abstraction and refinement are used multiple times until the system refines into the required abstraction level (i.e. detailed specification level).

D. Context_0

Context_0 designates the zero level context. It specifies the global level static parts of the model. The carrier sets specifies the static parts of the system. In CMS the carrier sets are of the offered courses, the degree programs, teachers, students, administration, computers, data, and records of all components of the system. The constants of a context do not change. The constants of the system as well as their description is presented below in table. The constants and carrier sets are used to write axioms, that are the most important part of a context.

Carrier Sets:	**Constants:**
COURSES PROGRAMS TEACHER STUDENT ADMIN COMPUTERS DATA RECORD	Admin /// Administrator of the CMS /// (all Privileges) Teacher /// Teacher access privileges Student /// Student access privileges max_c /// Maximum no. of courses that can be ///offered no_stds /// Number of Students min_tchrs /// Minimum number of Teachers no_cmptrs /// Number of Computers Courses /// Courses offered

The axioms are statements that are considered to be true. The elementary fundamental part of any model are the axioms. The machines use these axioms. The axioms of the CMS are defined as follows.

Axioms:
axm1:Admin ∈ ADMIN
axm2: Teacher ∈ TEACHER
axm3: Student ∈ STUDENT
// There are a finite number of users of CMS
axm4: finite(TEACHER)
// There are a finite number of programs offered in CMS
axm5: finite(PROGRAMS)
// There are a finite number of courses in all programs
axm6: finite(COURSES)
axm7: max_c ≤ card(COURSES)
// Minimum number of teachers is a Natural number
axm8: min_tchrs ∈ ℕ
// There is minimum 1:30 Teacher to Student ratio
axm9: min_tchrs > no_stds÷30
axm10: finite(COMPUTERS)
axm11: no_cmptrs = card(COMPUTERS)
axm12: Courses ∈ COURSES
axm13: finite(DATA)

E. Context_1

Context_1 is the refinement of the Context_0. It highlights those carrier sets and constants that are the extension of Context_0. Two new carrier sets i.e. USER and OBJECT are added into the system. The carrier set USER specifies the users of the CMS. A user can be administrator, teacher or student. The carrier set OBJECT specifies an object can be a data object or a classification object.

Extends:	**Carrier Sets:**	**Constants:**
Context_0	USER OBJECT	LEVEL

Axioms:
/// Each object is classified on a scale starting from 1 and ending
/// at 10.
/// The clearance level of each User of the system is also
/// classified on a scale between 1 and 10.
axm1: LEVEL = 1..10

F. Machine_0

A machine specifies and defines the dynamic properties (i.e. behavior) of the system. Machine_0 defines the first abstraction level that highlights the major functionalities of the proposed CMS.

1) Variables

Variables:	**Invariants:**
record student teacher admin	**inv1**: student ⊆ STUDENT **inv2**: record ∈ student → DATA **inv3**: teacher ⊆ TEACHER **inv4**: admin ⊆ ADMIN

G. Events

1) Add Student Record

It models the addition of a new student record into the CMS. When a student gets admission, the his/her record is added into the CMS.

INITIALISATION:	**Add_StudentRecord:**

Actions:
// Initial values are
// NULL
act1: record ≔∅
act2: student ≔∅
act3: teacher ≔∅
act4: admin ≔∅
	Any: st
	st_record
	Where: (Guards)
	grd1: st ∈ STUDENT
	grd2: st_record ∈ DATA
	grd3:
	record ∉ student → DATA
	Then: (Actions)
	act1:
	record ≔
	record ∪ {st ↦ st_record}

2) Check Student Record and Modify Student Record

The event Check_StudentRecord verifies if the student record is present in the CMS. The event Modify_StudentRecord changes or modifies the student record.

Check_StudentRecord:	**Modify_StudentRecord:**
Any: st st_record **Where: (Guards)** **grd1**: st ∈ STUDENT	**Any:** st st_data **Where: (Guards)** **grd1**: st ∈ dom(record)

ISSN – 1999-4974

grd2: st_record ⊆ record **grd3**: st_record ≠ ∅	**grd2**: st_data ∈ DATA **grd3**: st_data ∈∅ **Then:** **act1:** record ≔ record ⩤ {st ↦ st_data}

3) Who

Who:
Any: data
result
Where: (Guards)
grd1: data ∈ DATA
grd2: result = dom(record ▷ {data})

H. Machine_1

1) Variables

Variables:
object // Each object of the database
user // User of the Campus Management System (CMS)
odata // Each object in database has an object data component
class// Class of the user
clear// Clearance level of each user

2) Invariants

Invariants:
inv1: object ⊆ OBJECT
inv2: user ⊆ USER
inv3: odata ∈ object → DATA
inv4: class ∈ object → LEVEL
inv5: clear ∈ user → LEVEL

I. Events

1) Add User

INITIALIZATION: **Actions:** /// Initialization: /// Initial values are /// NULL **act1**: object ≔∅ **act2**: user ≔∅ **act3**: odata ≔∅ **act4**: class ≔∅ **act5:** clear ≔∅	**Add_User:** **Any:** u c **Where: (Guards)** **grd1**: u ∈ USER /// The new user must not already /// exist **grd2**: u ∉ user **grd3**: c ∈ LEVEL **grd4**: c ∈∅ **Then: (Actions)** /// The initial clearance level of the /// new user. **act1**: user ≔ user ∪ {u} **act2**: clear(u) ≔ c

2) Add Object

Add_Object:
Any: obj
data
cls
Where: (Guards)
grd1: obj ∈ OBJECT
/// The new object must not already exist
grd2: obj ∉ object
grd3: data ∈ DATA
grd4: cls ∈ LEVEL
grd5: odata ∈∅
Then: (Actions)
act1: object ≔ object ∪ {obj}
act2: odata(obj) ≔ data
act3: class(obj) ≔ cls

3) Read

Read:
Any: usr
obj
rslt
Where: (Guards)
grd1: usr ∈ user /// The user must exist
grd2: obj ∈ object /// The object must exist
/// A user can only read objects whose classification is less
/// than the user's clearance level.
grd3: clear(usr) ≥ class(obj)
/// The odata associated with the object
grd4: result = odata(obj)

4) Write

Write:
This operation overwrites the data value associated with the object with a new value.
Any: usr
obj
data
Where: (Guards)
grd1: usr ∈ USER
grd2: obj ∈ OBJECT
grd3: usr ∈∅
/// A user can only write objects whose classification less
/// than the user's clearance level
grd4: clear(usr) ≥ class(obj)
grd5: clear(usr) ∈∅
grd6: class(obj) ∈∅
grd7: data ∈ DATA
/// initially odata is empty
grd8: odata ∈∅
grd9: class(obj) ∉∅
Then: (Actions)
act1: odata(obj) ≔ d

5) Change class

Change_Class:
It ensures constraints on the user who is changing the object classification.
Any: obj
cls
usr
Where: (Guards)
grd1: obj ∈ object
grd2: cls ∈ LEVEL
grd3: cls ∈ ∅
/// A user can only write objects whose classification is less
/// than the user's clearance level.
grd4: clear(usr) ≥ class(obj)
grd5: clear(usr) ≥ cls
grd6: class ∈ ∅
Then: (Actions)
act1: class(obj) ≔ cls

ISSN – 1999-4974

6) Change Clear

Change_Clear:
This event provides constraints on the user who is changing the object classification.
Any: usr
any
cls
Where: (Guards)
grd1: usr ∈ USER
grd2: any ∈ USER
grd3: any ∈ ∅
grd4: clear(any) ≥ clear(usr)
grd5: clear(any) ≥ cls
grd6: cls ∈ LEVEL
grd7: clear ∈ ∅
Then: (Actions)
act1: clear(usr) ≔ cls

7) Remove User

Remove_User:
Any: usr
Where: (Guards)
grd1: usr ∈ USER
grd2: clear ∈∅
Then: (Actions)
act1: user ≔ user \ {usr}
act2: clear ≔ {usr} ⩤ clear

8) Remove Object

Remove_Object:
Any: obj
Where: (Guards)
grd1: obj ∈ object
grd2: odata ∈ ∅
Then: (Actions)
act1: user ≔ user \ {usr}
act2: clear ≔ {usr} ⩤ clear
act3: odata ≔ {obj} ⩤ odata

VI. CONTRIBUTIONS

In this work our contributions are:

- A development approach based on the fundamental software engineering principle of correct-by-construction, following the underlying principles of abstraction and refinement.
- The modeling and formal proving of a formal model of the CMS.
- The mathematical proofs of the correctness properties of the CMS.
- Periodic refinement and abstractions of the CMS. Starting from an abstract level and ending into a detailed refined model. This model can further be refined in the form of a number of refinement layers.

VII. CONCLUSION AND FUTURE WORK

The formal modeling and proving of CMS allows exhaustive investigation of the system properties. A complete exhaustive formal model of the CMS has a number of abstraction layers, starting from very abstract concepts that are step-wise refined into detailed concrete concepts. This detail model presents an accurate, precise, exhaustive and formally correct model of the system. In this paper the proposed CMS has two abstraction layers i.e. layer-zero and layer-one. The future work is the automated generation of UML-B diagrams of the proposed CMS. These diagrams would provide an exhaustive state space graphs showing all possible states as well as deadlock states. Methodologies consisting of formal modeling and implementation exhaustively specify the system and therefore subsequently ensure high levels of correctness.

REFERENCES

[1] Jean-Raymond Abrial, Modeling in Event-B System and Software Engineering.: Cambridge University Press, 2010.

[2] Jean-Raymond Abrial et al., "Rodin: An Open Toolset for Modelling and Reasoning in Event-B," International Journal on Software Tools for Technology Transfer (STTT), vol. 12, no. 06, pp. 447-466, November 2010.

[3] Lili Xu and Hong Zhang, "Formal Verification of Software Safety Criteria Using Event-B," in International Conference on Reliability, Maintainability and Safety (ICRMS), 2014, pp. 342-347.

[4] Fabio Levy Siqueira, Thiago C. de Sousa, and Paulo S. Muniz Silva, "Using BDD and SBVR to refine business goals into an Event-B model: a research idea," in IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2017, pp. 31-36.

[5] Dieu Huong Vu, Anh Hoang Truong, Yuki Chiba, and Toshiaki Aoki, "Automated testing reactive systems from Event-B model," in 4th NAFOSTED Conference on Information and Computer Science, 2017, pp. 207-212.

[6] Yousra Ben Daly Hlaoui, Ahlem Ben Younes, and Leila Jemni Ben Ayed, "From Sequence Diagrams to Event B: A Specification and Verification Approach of Flexible Workflow Applications of Cloud Services Based on Meta-model Transformation," in IEEE 41st Annual Computer Software and Applications Conference, 2017, pp. 187-192.

[7] Nestor Catano, "An Empirical Study on Teaching Formal Methods to Millennials," in IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials (SECM), 2017, pp. 3-8.

[8] Guillaume Babin, Yamine Ait-Ameur, and Marc Pantel, "Web Service Compensation at Runtime: Formal Modeling and Verification Using the Event-B Refinement and Proof Based Formal Method," IEEE TRANSACTIONS ON SERVICES COMPUTING, vol. 10, no. 1, pp. 107-120, January/February 2017.

[9] Michael Jastram and Michael Butler, Rodin User's Handbook: Covers Rodin V.2.8. USA: CreateSpace Independent Publishing Platform, 2014.

[10] M. Leuschel and M. Butler, "ProB: A Model Checker for B," in Proceedings FME 2003, Pisa, Italy, 2003, pp. 855-874.

 ISSN – 1999-4974