

Building Environment Rule and Analysis (BERA) Language

And its Application for Evaluating Building Circulation and Spatial Program

A Dissertation
Presented to
The Academic Faculty

by

Jin Kook Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Architecture

Georgia Institute of Technology
May, 2011

COPYRIGHT 2010 BY JIN KOOK LEE

Building Environment Rule and Analysis (BERA) Language

And its Application for Evaluating Building Circulation and Spatial Program

Approved by:

Professor Charles M. Eastman, Advisor
College of Architecture and Computing
Georgia Institute of Technology

Dr. James H. Garrett
College of Engineering
Carnegie Mellon University

Dr. John Peponis
College of Architecture
Georgia Institute of Technology

Dr. Jochen Teizer
College of Engineering
Georgia Institute of Technology

Dr. Shamkant B. Navathe
College of Computing
Georgia Institute of Technology

Date Approved: December 9, 2010

BIM – Building Information Modeling – is a process.

Chuck Eastman

ACKNOWLEDGEMENTS

This dissertation would not have been written without the support and feedback of many people. I could not even get started this study if there was no guidance of Professor Chuck Eastman to this intriguing and challengeable topic. I would like to thank him first, my advisor and mentor for my PhD studies in the Design Computing Program at Georgia Tech. His insight on the research realm of design computing was very essential for the research and development addressed in the dissertation. I am very proud of the fact that a huge challengeable research area – Building Information Modeling (BIM) and related work is one of my research directions that have been explored for my PhD studies advised by him, as people call him “The Father of BIM”.

Thanks to my thesis committee professors for helping me structure and further develop my researches. I am much honored to have very renowned scholars who have expertise in design computation, spatial analysis, databases in computer science, and civil and environmental engineering as my thesis committee members. Professor John Peponis in Organizational & Cognitive Performance Program in Georgia Tech College of Architecture is the coordinator of Post-Professional Program. His papers and theoretical approaches in spatial analysis and cognition broadened my view towards developing my thesis. Professor Sham Navathe in Georgia Tech School of Computer Science was my minor advisor. His classes on the fundamental database theories and object-oriented modeling approaches helped me clarify conceptual modeling issues. Professor James Garreett, as an external reader, is the head of Civil and Environmental Engineering at Carnegie Mellon University. He encouraged me to keep researching and broadening the further research perspectives, as well as another renowned external reader Professor Jochen Teizer in Georgia Tech College of Engineering. Their advices on the thesis are also very helpful for the further directions of the research.

There are several people that I owe acknowledgements for giving me thoughtful and sincere comments on my work. I have been involved in a project to develop a set of automated building design checking modules for the US Courthouses funded by the US GSA. It was called the GSA Courthouse Design Guide Automation project. The Georgia Tech project team including the author, led by Professor Chuck Eastman, has developed several design checking software modules based on SMC (Solibri Model Checker) including building circulation and security checking, space program review, preliminary energy analysis, and cost estimate. Through the GSA project, my advisor Chuck Eastman offered us to develop a new language-driven method for the building environment rule and analysis (BERA). The BERA Language was named and initiated by his idea in the early phase of the GSA project. I am indebted to the GSA project P.I. Chuck Eastman and members for their support, help, and collaboration. I am also thankful to Fred Miller, Calvin Kam and Peggy Yee at GSA; Pasi Paasiala and Matti Kannala at Solibri Oy, Finland; GSA project members at Georgia Tech; and all who have been involved in the project.

While I was working at a housing design and manufacturing company Hanssem Co. Ltd. in Korea in 2000, the year just after leaving undergrad school, I did not know that Design Computing is my field of study for my Ph.D. Much thanks to a guidance of Mijeong Kim, one of my alma mater seniors at Yonsei University who is now a professor at Kyunghee University, I was involved in graduate level researches and projects emphasis in Digital Design Media. I am very grateful to Professor Hyunsoo Lee, my former advisor for the M.S. at Yonsei University. He always encouraged me to keep motivated whenever I was trying to do something in the design and computing domain. I am also thankful to Yeunsook Lee, Mikyung Ha, Jinwon Choi, the professors at Yonsei University who spoke highly of me to do further studies. I think I was a lucky graduate student to be advised by such renowned and warm-hearted Professors at Yonsei University. I am always proud of it.

I thank the PhD Program at Georgia Tech College of Architecture. It was extremely lucky event in my life when I have got admitted to Georgia Tech as a PhD student, because the combination of Professor Chuck Eastman, Design Computing, and Georgia Tech was my only one application for the PhD program at that time. If there was no admission from Georgia Tech (thankfully, with full-funding), probably I would be just a designer, developer, or programmer in a certain company in Korea – without PhD studies. Again, it is thanks to my advisor Chuck Eastman. He gave me the chance even though I was not the typical best applicant among a lot of PhD applicants. I have tried to prove that his decision was right. The person who I used to be yesterday was my biggest competitor throughout my PhD studies.

I would not have been able to enjoy my life at Georgia Tech without good friendship. I thank Sherif Abdelmohsen, Hugo Sheward, Paola Sanguinetti, Andres Cavieres, Marcelo Bernal, Matthew Swarts, Frank Wang, Ghang Lee, Donghoon Yang, Seokjoon You, Jaemin Lee, Yeonsuk Jeong, Jinsol Kim, Junha Kim, Youjong Cheong, Yeonsook Heo, Sanghoon Lee, Jaeho Yoon, Jihyun Kim, Hyunbo Seo, Seunghyun Lee, and all others for their feedback and friendship. I am also thankful to Professor Edoo Kim who was a visiting scholar from Ulsan University.

I thank my parents for raising me an optimistic and patient person. Thanks to my sister Soojung, Juhyung and their families for being my constant supporters. Thanks to my parents-in-law for their warm-hearted support for me and my wife. It is not easy to thank enough to the family.

Lastly, I cannot thank enough my wife, Eun Joo Kim for her patience and support. I thank for her ongoing support and dedication as always. This dissertation is for Eun Joo and my daughter. Lori, my adorable daughter, makes me happier than ever before. My family Eun Joo and Lori has always been the motivation of my life.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	xi
LIST OF FIGURES	xii
LIST OF ABBREVIATIONS	xvi
SUMMARY	xvii
<u>CHAPTER</u>	
1 INTRODUCTION	1
1.1. Introduction	1
1.1.1. What is BERA Language	1
1.1.2. The Purpose of Developing BERA Language	5
1.1.3. Domain, Scope and Primary Goal	9
1.1.4. Research Questions Regarding Language Design and Implementation	12
2 BACKGROUND	16
2.1. Background	16
2.1.1. Overview	16
2.1.2. The Problem	17
2.2. Domain-specific Languages within BIM	17
2.2.1. IFC as a BIM Domain-specific Modeling Language	17
2.2.2. BIM Domain-specific Programming Languages	18

2.3. Lessons Learned from Reviewing Other Languages	22
2.3.1. Scripting Languages Based on ECMA Script	23
2.3.2. Domain-specific Languages Based on Java	26
3 BERA LANGUAGE DESIGN	30
3.1. Design Strategy	30
3.1.1. Overview	30
3.1.2. BERA Language Design Strategy	31
3.2. Abstraction of the BERA: BERA Object Model	34
3.2.1. Overview	34
3.2.2. Building Environment Rules	35
3.2.3. Building Model-centered Abstraction of the Building	38
3.2.4. BERA User-centered Abstraction of the Building	39
3.3. BERA Language Design	43
3.3.1. Lexical Design	43
3.3.2. Syntactic Design	44
4 BERA LANGUAGE DEFINITION	47
4.1. Overview	47
4.2. Reference Declaration	51
4.3. BERA Object Model Declaration	54
4.3.1. BERA Object Model	54
4.3.2. Space and SpaceGroup Object	56
4.3.3. Object Properties	62
4.3.4. Data Value and Operation	72

4.3.5. Quantification and Conjunction for the BOM Expressions	74
4.4. Rule Definition Statement	76
4.5. BERA Execution Statement	78
 5 BERA LANGUAGE IMPLEMENTATION	 82
5.1. Implementation Overview	82
5.2. BERA Listener	85
5.2.1. Lexical and Syntactic Analysis	85
5.2.2. Semantic Analysis	87
5.3. BERA Object Model Handler	91
5.3.1. Static BOM Builder	92
5.3.2. Dynamic BOM Builder	93
5.4. BERA Executor	96
5.5. BERA Language Tool	98
5.6. BERA Language Extensibility	101
5.6.1. Re-targetable BERA Language	102
5.6.2. Extensible BERA Object Model	105
 6 APPLICATIONS AND EVALUATIONS	 108
6.1. Overview	108
6.2. Real-world Rules and Analysis	110
6.2.1. Software-driven Methods for Handling Real-world Rules	110
6.2.2. Handling Rules by the BERA Language	112
6.3. Applications for Handling Building Objects	115
6.3.1. Static Building Elements	116

6.3.2. Static Meta-Element Example: Graph	120
6.3.3. Dynamic Object Group Definition: SpaceGroup	123
6.3.4. Floor Definition	126
6.3.5. Path Definition	128
6.4. Application for Evaluating Spatial Program	133
6.4.1. SpaceGroup Rule	133
6.4.2. Floor Rule	139
6.5. Application for Evaluating Building Circulation	143
6.5.1. Circulation Rule	143
6.5.2. BERA and Target Language-based Execution	148
6.6. Application for other Building Objects	156
6.7. Evaluation of BERA Language	159
 7 CONCLUSION	 161
 APPENDIX A: EBNF Notation as a Context-free Grammar Definition	 165
APPENDIX B: ANTLR as a Tool for Defining Domain-specific Language	166
APPENDIX C: BERA Language Grammar	167
APPENDIX D: BERA Language User Manual	184
REFERENCES	185
VITA	197

LIST OF TABLES

	Page
Table 4.1. Building properties and description.	64
Table 4.2. Floor properties and description.	68
Table 4.3. Space properties and description.	69
Table 4.4. SpaceGroup properties and description.	70
Table 4.5. Path properties and description.	71
Table 5.1. Some valid examples of the dot-notation access to BOM (examples of DOTExpr in Figure 5.4). In the example expression column, words starting with upper case letters are BOM names, and words starting with lower cases are property names or quantification words. User-defined variable names for dynamic BOM are in italic.	89
Table 6.1. To retrieve static building objects using: <code>get (args)</code> .	117
Table 6.2. To retrieve static graph elements using: <code>get (args)</code> .	121
Table 6.3. To instantiate a certain group of space objects named ‘midOffice’ using a typical dynamic BOM definition method. It can be simply retrieved by ‘get’ command with ‘midOffice’ as its argument.	124
Table 6.4. To instantiate two user-defined floors named ‘bigFloor’ and ‘upperFloor’ and display them using get command. They are visualized in the system using their boundary polygon, and displayed in the BERA console with associated information in current BERA Language Tool.	126
Table 6.5. To instantiate two different SpaceGroup ‘start’ and ‘end’ first, and put them into another dynamic BOM (Path) definition named ‘myPath’. The BERA Language Tool computes circulation paths between two space groups. As shown in this Table, ‘start’ has 3 spaces and ‘end’ contains 4 space instances, therefore total 12 circulation path instances are populated by this definition. The system visualizes all of their paths with graph structures and highlighted start and end spaces.	129
Table 6.6. To evaluate a rule named ‘officeRule’ using input space group named ‘research’. The entire set of ‘research’ is displayed in the BERA console as below, and which ones are passed or not also printed out in the console area. Three passed space objects are highlighted in 3D visualization. But only one space object is passed in the rule checking for ‘officeRule2’ that is inherited from ‘officeRule’.	134

- Table 6.7. To instantiate two floor objects named ‘floor1’ and ‘floor2’, and put those into a user-defined rule named ‘floorRule’ which regulates certain floor conditions. This example evaluates two checking instances using two input floor parameters. One is passed and another is failed as described in this table. 139
- Table 6.8. To instantiate a circulation rule named ‘myOfficeCircRule’ and evaluate it using two pre-defined BOM named ‘visitorOffices’ and ‘meetingSpaces’. The execution statement is simply its rule name with two arguments as follows. All 14 path instances are visualized in SMC, and the BERA console displays their rule checking result in detail. 143
- Table 6.9. To instantiate circulation paths from “visitors conference” to all the spaces in the same floor – in this example, 92 path instances are populated. Not only is its rule checking result, but also an advanced Java program result also displayed in the BERA console. 149
- Table 6.10. To evaluate a rule named ‘externalWallRule’ using input object group named ‘exWalls’ which contains Wall object instances. The associated information for ‘my’ is displayed in the BERA console as below, and which ones are passed or not also printed out in the console area. Three passed Wall objects are visualized in 3D. 157

LIST OF FIGURES

	Page
Figure 1.1: The location of BERA Language in a high-level classification of computer languages. BERA is a domain-specific programming language.	3
Figure 1.2. Two Evolutions of building design review workflow. The second evolution introduces the purpose of this study: automated design reviews enhanced by BERA Language.	8
Figure 1.3. The building objects, their properties and relations within BERA. The BERA Language design aims to be open-ended. This dissertation will demonstrate its applications for evaluating the general domain of building circulation and spatial programming rules.	11
Figure 2.1. A screenshot of the 3D Spatial Query Language prototype application [Borrmann, 2010a].	21
Figure 2.2. Example of the target objects of JavaScript: Document Object Model (DOM).	25
Figure 2.3. A simplified example of an execution of Processing language.	28
Figure 3.1. A Simplified IFC Entity Hierarchical Structure for the Building Elements.	39
Figure 3.2. An Abstraction of the Building Model for BERA Users (BERA-centered).	40
Figure 3.3. BERA Object Model design, within the scope of this study.	42
Figure 4.1. BERA Object Model within the scope of this study: spatial objects. A circulation path or a fire egress path is also another type of Space object, as a dynamically instantiated object.	56
Figure 4.2. Spatial objects and their properties: statically instantiatable BERA Object Model (BOM).	64
Figure 4.3. Spatial objects and their properties: dynamically instantiatable BERA Object Model (BOM).	65
Figure 4.4. A circulation path, as one type of a dynamically instantiated SpaceGroup object, can be represented by a set of space objects, and it can be linked with other type of data structure such as a graph structure implemented by an additional BERA library for it.	66

Figure 5.1. High-level Execution Architecture of the BERA Language. As a high-level data flow diagram, arrows mean data flow, and up/down arrows also include interactions.	83
Figure 5.2. BERA Language Architecture Detail: Front-end and Back-end.	84
Figure 5.3. An overview diagram to describe the top-level lexical and syntactic analysis of BERA Language input program.	86
Figure 5.4. An overview diagram to describe establishing the BERA Language semantics.	88
Figure 5.5. An example of the BERA parser implementation and its data structure.	90
Figure 5.6. BERA Object Model classes overview.	92
Figure 5.7. Overview of the dynamic BOM and its properties implementation.	94
Figure 5.8. Object selection algorithm overview: multiple conditions DefCond in a DefBOM iterate Object Collection and collect its array of Boolean results.	95
Figure 5.9. Space object selection example for the above program: an instance <i>Space[3]</i> is selected because its result is true. A Boolean array {T, T, T, F} returns T because one of the conjunctions is “or” and its value is T. (Left to right evaluation)	96
Figure 5.10. Path object selection example: an instance <i>Path[3]</i> is passed because its result is true. A Boolean array {T, F, T, T} returns T because one of the conjunctions is “or” and its value is T. (Left to right evaluation)	98
Figure 5.11. BERA Language Tool start-up screen.	99
Figure 5.12. BERA Language Tool interface.	99
Figure 5.13. BERA Language Tool on top of the BIM platform – SMC.	100
Figure 5.14. A brief data flow diagram to describe the implementation of the BERA Language Tool. A BERA code is translated into a Java code and executed.	102
Figure 5.15. Overview of the Re-targetable BERA Language Translator.	104
Figure 5.16. An example of extended building objects: <i>Structure</i> is an example class of the dynamic BOM of these structural building objects. For example, the object group “all walls and slabs of the basement floor” or “all exterior walls” can be dynamically instantiated as an instance of <i>Structure</i> , by the user. It will be used for the rule checking, analysis, or just for various visualizations.	106
Figure 5.17. Two-way extensibility on the BOM.	107

Figure 6.1. Table-based parameters example for the US Courthouse circulation rules.	110
Figure 6.2. Table-based parameters example for one of the conditions shown in Figure 6.1.	112
Figure 6.3. Table-based parameters example for collecting space object components using parameters.	112
Figure 6.4. A test model for running BERA Language programs in this chapter.	116

LIST OF ABBREVIATIONS

AEC	Architecture, Engineering, and Construction
AECFM	Architecture, Engineering, Construction, and Facility Management
BERA	Building Environment Rule and Analysis
BIM	Building Information Modeling
BOM	BERA Object Model
CIS/2	CIMSteel Integration Standards Release 2
DSL	Domain-Specific Language
IDE	Integrated Development Environment
IFC	Industry Foundation Classes
JVM	Java Virtual Machine
SMC	Solibri Model Checker®
STEP	Standard for the Exchange of Product Model Data
SQL	Structured Query Language
W3C	World Wide Web Consortium
AIA	American Institute of Architects
ADA	Americans with Disabilities Act
ADAAG	ADA Accessibility Guidelines for Buildings and Facilities
GSA	The US General Services Administration
NFPA	National Fire Protection Association
OSHA	Occupational Safety and Health Administration
USCDG	U.S. Courts Design Guides

SUMMARY

This study aims to design and implement a domain-specific computer programming language: the Building Environment Rule and Analysis (BERA) Language. As a result of the growing area of Building Information Modeling (BIM), there has been a need to develop highly customized domain-specific languages for handling issues in building models in the architecture, engineering and construction (AEC) industry sector. The BERA Language attempts to deal with building information models in an intuitive way in order to define and analyze rules even in early design stages. The application of the BERA Language aims to provide efficiency in defining, analyzing and checking rules. Specific example applications implemented in this dissertation are on the evaluation of two key aspects: building circulation and spatial programming. The ultimate goal of BERA Language is, however, potentially to cover the broad range of applications for the AEC. Thus, this dissertation attempts to describe the language design issues regarding its potential extensibility.

The objective of this study is to develop a rule checking language architecture supporting ease of use, high fidelity, extensibility and portability, and consequently to design and implement a high-level and domain-specific language: BERA Language. The goal is to accomplish an effectiveness and ease of use without precise knowledge of general-purpose languages that are conventionally used in BIM software development. To achieve these goals, this study proposes an abstraction of the universe of discourse - it is the BERA Object Model (BOM). The design and implementation of the BERA Language focuses on building objects and their associated information-rich properties and relationships. Especially, most of spatial information, which is the main subject of this implementation, is derived and computed from the spatial data defined in three-

dimensional BIM models, rather than the two-dimensional footprints of building elements such as walls and slabs.

This dissertation consists of two main parts: 1) description of the design and formal definition of the BERA Language, and 2) implementation of the BERA Language Tool and its application. The former part attempts to answer the research question, involving the effectiveness, ease of use, and extensibility of the language for end users. This front-end part is standard for all other implementations. The latter part is a practical and technical guideline for the actual development of the implementation. Implementation issues are mostly related to the building information models, their mapping into the BOM structure, and their instantiation and execution by the language. Portability of the language and platform-dependent issues are also involved in the BERA Language Tool implementation. This latter back-end part varies by implementation environments. The implementation of this study is based on the use of Industry Foundation Classes (IFC) as given building information models, Solibri Model Checker® (SMC) as an IFC engine, and the Java Virtual Machine (JVM) as a compilation and execution environment.

The proposed BERA Object Model (BOM) is a human-centered abstraction of complex state of building models rather than the computation-oriented abstraction which is generally intended to cover broad-ranged issues. BOM is one of the key concepts to the building environment rule and analysis as the language name literally implies. By using BOM, users can enjoy the ease of use and portability of pre-defined BIM data, rather than complex and platform-dependent data structures. A newly proposed BOM data structure operated on building objects, focusing on evaluating building circulation and spatial program within the scope of this research, but this study also has reviewed and demonstrated its potential for extensibility. The author realized that it is another challenge to define generic and valuable BOM as it grows more detailed. Not only its lateral extensions such as structural building elements, but also the vertical extensions

such as additional properties for existing BOM objects are good examples of its extensibility. In the BERA Language Tool implementation described in this dissertation, many computed and derived properties have been proposed and implemented for the building environment rule and analysis (BERA), as well as some basic data directly from the given building model.

The BERA Language Tool is an integrated development environment for the proposed BERA Language. By using BERA Language Tool, users can evaluate their programs on their building models, focusing on both design analysis and rule checking of the purposes of building circulation and spatial programming. The proposed tool is an example implementation developed by the author. Substantial benefits can be taken from using BERA Language, and they can be summarized as follows:

- 1) Ease of use: Contrary to the general-purpose languages, the BERA Language is easy to use for domain experts, but still powerful to handle domain-specific problems.
- 2) Extensibility: the BERA Language offers an open-ended model for the human-centered abstraction of a building – BOM.
- 3) Portability: BERA Language can be embedded in several other types of BIM applications such as BIM authoring tools, with consistent front-end features.

Target users of the BERA Language are domain experts such as architects, designers, reviewers, owners, managers, students, etc., rather than BIM software developers. It means that the people who are interested in the building environment rule and analysis are the potential users. The rules implemented and applied in this study involve building circulation and spatial programming. Building circulation and spatial programming are two of the crucial topics in conceptual design stages of the building project. This study presents pragmatic applications that define rules for these topics and evaluate them using the BERA Language Tool. This tool comprises many libraries to

alleviate common but unnecessary problems and limitations that are encountered when users attempt to analyze and evaluate building models using commercially available tools. Combined with other additional libraries which populate rich datasets for certain purposes, the BERA Language will be fairly versatile to define rules and analyze various building environmental conditions.

CHAPTER 1

INTRODUCTION

1.1. Introduction

1.1.1. What is BERA Language

As a result of the growing area of Building Information Modeling (BIM), there has been a need to develop highly domain-customized computer languages for handling specific issues in building models in the architecture, engineering and construction (AEC) industry sector¹. The BERA (Building Environment Rule and Analysis) Language attempts to deal with building information models in an intuitive way in order to define and analyze rules that can be applied to check the building objects configuration, composition and layout of buildings, even in the early stages of design. The intended use of the BERA Language is to analyze and assess rules regarding building environment as BERA literally means. This research aims to design BERA Language within its intended use, and attempts to demonstrate its implementations focusing essentially on the evaluation of two main aspects: building circulation and spatial programming.

¹ A software development project team including the author, funded by the US GSA and led by Prof Eastman, has developed several automated building review and rule checking modules including building circulation checking, space program review, energy analysis and cost estimate since 2006 [GSA-GT, 2010; GSA Project - dcom.arch.gatech.edu/gsa]. They are Solibri Model Checker (SMC)-based plugin software. The team realized that there was a need to develop language-driven methods for building design rule checking, while these software-driven (parameter-driven) methods have been developed by the team. See Figure 1.2 and the section 6.2 for comparing those two approaches.

The BERA Language is developed for buildings that are represented in various building information models populated by most BIM authoring tools with either proprietary or public schemas. Proprietary schemas include commercial and native formats that are commonly used in the field such as Revit® models [Autodesk, 2010], ArchiCAD® models [Graphisoft, 2010], Bentley Architecture® models [Bentley Systems, 2010], and so on. Public schemas include open and neutral formats such as the Industry Foundation Classes (IFC) and CIS/2 [AISC, 2010; buildingSMART, 2010a; Yang D et al, 2010]. IFC is written in EXPRESS [ISO 10303, 1994; buildingSMART, 2010b] which supported development of one of the building *modeling languages*. The BERA Language is designed to become one of the *programming languages* for handling these building models. Figure 1.1 illustrates where the BERA Language and IFC are located within a high-level classification of computer languages: along the modeling languages [Booch, 1998] - programming languages [Scott, 2005; TIOBE Soft, 2010] axis, and the general-purpose languages - domain-specific languages (DSL) axis². The BERA Language is a domain-specific programming language for various building information models. The initial implementation of BERA involves however IFC public BIM schema. Future implementations of BERA-like languages are expected to involve proprietary BIM platforms. Most BIM authoring tools have their own, mostly internally hidden, extensible semantics, based on extensible properties and relations. A building model schema and operations will thus in some cases have to be extended to support particular rules.

² The classification of computer languages is always controversial. There are several factors to classify them – purpose, type identifier, platform, target user, origin, syntax style, object-oriented concept, and so on. Figure 1.1 does not attempt to define a new classification of computer languages. It attempts to introduce the location of BERA Language and IFC using high-level and simplified two axes.

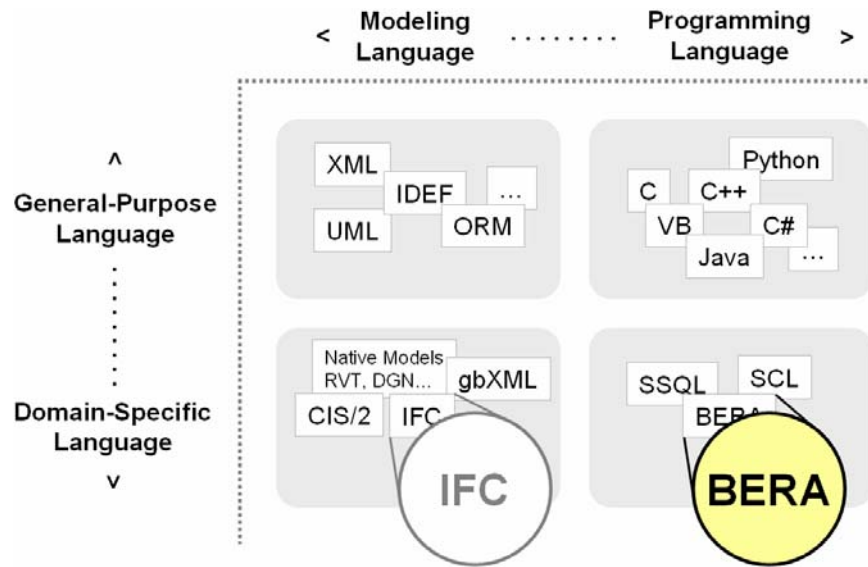


Figure 1.1. The location of BERA Language in a high-level classification of computer languages. BERA is a domain-specific programming language.

The current version of IFC is 2x3TC1 (2x Edition 3 Technical Corrigendum 1) [IFC 2x3TC1, 2010] as of 2010. Its universe of discourse is a subset of all the entities defined in IFC. [buildingSMART, 2010a] By implication, it is not a closed definition. As new entities are added to IFC, they can potentially be addressed within the BERA Language. However, this addition is not automatic. The subset of IFC that is the domain of discourse must be explicitly defined. The universe of discourse for an assessment program written in BERA will vary according to its building type and the checks written for that building type. The relevant checks will thus have an associated IFC View that it will require.

Regarding the initial target applications in this dissertation, the BERA Language addresses issues of spatial layout and composition, applied to both single spaces and large sets of spaces and their composition. This scope is in contrast to the more general one for building materials and construction methods, which is the universe of discourse for the majority of building codes, general design guides, specific requirements, and so on. In general, spatial design rules are best practices, and most of the best practices are written

in the form of design guide or regulations such as Hospital Design Guide [AIA, 1997], U.S. Courts Design Guide [USCDG, 1997; 2007], and so on. There is another type of rules within a specific binding code or regulation. They contain the Fire Code [NFPA, 2010; OSHA, 2010] that regulate certain egress path-related conditions in case of fire, and the ADA related codes from Accessible Design Guide [US Access Board, 2004], etc. These documents contain the “rules” for review, and the BERA Language attempts to deal with them. The scope and the domain of this language application is specifically focusing on the architectural considerations of building circulation and spatial programming based on space objects, their associated properties, and relations. In general, these are valuable resources for design evaluation tasks even in early phase of design, and the design evaluations are very important determinant factors of success in building projects. Early evaluation of building design increases the overall performance and quality of the building [Eastman et al, 2008].

This study aims to design and implement the BERA Language, which can be summarized as follows:

- 1) The BERA Language can be classified along high-level computer language categories as follows:
 - a domain-specific language,
 - a programming language, and
 - a rule language.
- 2) Its main functionality is the definition and analysis of building design rules, those related to any building objects. In terms of the scope of applications, especially spatial objects can be classified as follows:
 - space objects and diverse kind of group of spaces,
 - spatial properties derived and computed from the spaces and space groups, and
 - relations of spaces, including topological connections, metric distances, etc.

- 3) Its target field is basically the AEC (architecture, engineering, and construction) industry, especially focusing on design analysis and rule checking issues in conceptual design phases of building projects.
- 4) It does not attempt to modify and change building designs directly, because it aims to analyze and check a building design model, as BERA literally means.

1.1.2. The Purpose of Developing BERA Language

In the following we introduce an example of domain-specific languages (DSL) in a different domain. Recent database management systems (DBMS) provide very powerful GUI (Graphic User Interfaces) for managing databases. If a certain DBMS does not provide an SQL (Structured Query Language) [SQL, 2010] command prompt interface, that is, providing only pre-defined GUI without SQL text-based data manipulation interface, it will be weeded out from the market. GUI and API are optional, but SQL is fundamental to running databases in most DBMS [Khan, 2006]. People in charge of managing databases are familiar with using SQL simply because of its simple yet powerful capability. Databases have become an essential component of everyday life in modern society [Elmasri and Navathe, 2005]. Consequently the specific programming language SQL has also been broadly used by many people in diverse fields. SQL was essentially one of the domain-specific languages since its emergence in the 1970's [Chamberlin et al, 1974; SQL, 2010], but nowadays it is no more known just as “domain-specific language” because it is everywhere. This example shows how a well-designed small language can be a bigger one that is capable of managing problems in a specific software framework.

In the AEC field, architects design a building and produce various building model files even in the early concept design phase. This early concept design is a significant determinant of the eventual success and impact of the project [Eastman et al, 2009a]. It is thus important to validate the design and update it within a reasonable timeframe. Before BIM, the only means to deal with the complex knowledge in building design was through a manual and intuitive review process. The advent of BIM applications allowed for the parametric generation of designs that respond to various criteria, the prospect of computer-interpretable models and the automated checking of designs after they are generated [Eastman et al, 2008]. IFC is commonly used to perform a recursive reviewing process on building models because it is a design tool-independent and neutral data model representation supported by most BIM design tools. It is being used as an interoperable building model representation in most recent BIM-enabled automated building design reviewing and checking efforts [Eastman et al, 2009b]. Domain experts such as design reviewers, code supervisors, or building owners review the IFC using a set of tools developed by software developers. Figure 1.2 depicts this workflow of building model review tasks in a simplified transitional diagram. In this workflow, the reviewers' workbench is moved into the BIM software that is a set of applications which support multiple functions for importing IFC, visualizing the model, reviewing/checking certain features of the design such as building program and circulation, and so on. Solibri Model Checker® [Solibri, 2010] is one of the commercial tools that were developed for domain experts who are in charge of reviewing building models. Developers constantly upgrade their tools, user interfaces, and sometimes plug-ins responding to user feedback. However, this is still a software-driven method (in other words, a parameter-driven method), where user review tasks are limited within the pre-defined capabilities that are provided by the BIM software.

The conventional way of doing a design review is manual by a person interpreting the design rules and applying them from their knowledge, and recently some BIM-

enabled software-driven methods have been developed (See Figure 1.2). One of the ideal ways to accomplish an effective and advanced design review task is simply to provide an easy and effective high-level language for handling specific problems for domain experts. Without a language-based interaction with building models, domain experts do their job within the scope of the pre-defined BIM software functionalities. If more functionality is required, domain experts should use another application, or develop their own customized application as an add-on or plug-in. In other words, the building review tasks by the software-driven methods are dependent on the pre-defined functionality of the tools, and the process of problem solving in developing new functionalities is time-consuming. For example, we have developed a circulation checking module on top of the Solibri Model Checker® (SMC) interface [Eastman et al, 2009b; Lee J-K et al, 2010; Lee J et al, 2010]. It supports the assignment of any start spaces and target spaces for retrieving possible routes between two given spaces. SMC checks the results as the rules are defined using a parameterized table. This table contains pre-defined properties of circulation paths such as security level, metric distance, direct adjacency, and so on. This is considered a powerful method for checking building circulation in its pre-defined scope. However, in cases where a domain expert needs to check another property, or if he/she needs to insert a logical checking such as IF- THEN- ELSE-, there is no way to realize these instantly without changes in the software. This is similar in the case of cross-criteria checking; e.g. IF A is passed, THEN check B, where A and B are in different rule criteria. The limitations from this specific software-based design review framework can be summarized as follows:

- 1) The software-driven method offers only pre-determined and limited functionalities. When a new type of parameter is required, this pre-defined parameter-based method cannot support it.
- 2) There is no way to review complex rules beyond the pre-defined coverage. For example, cross-criteria rules, conditional and logical rules, etc.

3) It is available only within the scope of pre-defined functions and rule types.

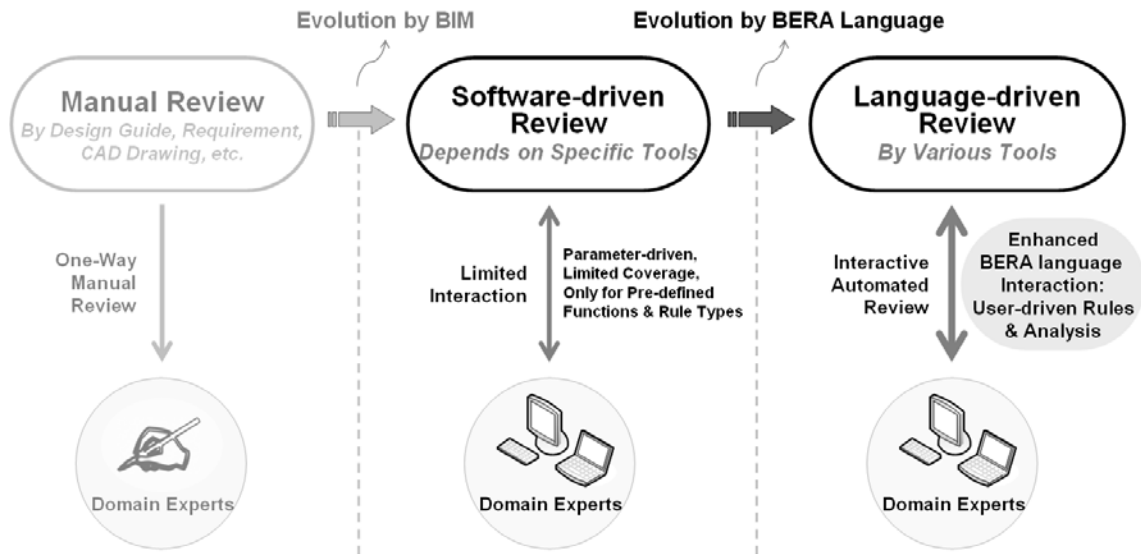


Figure 1.2. Two Evolutions of building design review workflow. The second evolution introduces the purpose of this study: automated design reviews enhanced by BERA Language.

The BERA Language is important specifically at this point. Domain specific languages are important to develop software because they represent a more natural, high fidelity, robust, and maintainable means of encoding a problem than simply writing software in a general-purpose language [Metsker, 2001; Parr, 2008]. The second evolution in Figure 1.2 simply introduces an advanced workflow enhanced by the BERA Language.

Followings are the purposes and benefits of using the BERA Language:

- 1) The BERA Language is basically a rule checking language that enables review of building models. Concept design review is a significant determinant of the eventual success and impact of a building project. Using the BERA Language is thus beneficial for the decision making process in this early phase of building projects in an easier and faster way.

- 2) Contrary to the general-purpose languages, the BERA Language is easy to use for novice or non-programmers. This means that domain experts such as architects or reviewers can obtain much more control and better handling of building models.
- 3) It is easier than general purpose languages, but still powerful to handle domain-specific problems. The BERA Language supports various operational statements for handling both the complexity of design rules, and the complex relations of space objects and properties. They are logic operations, logic values, recursion, negation, inheritance, polymorphism, algebraic operations, and so on. These are fundamental constructs of the general-purpose language, and the BERA Language is also taking advantage of it.
- 4) In terms of its extensibility, the BERA Language offers an open-ended model for the abstraction of a building. (Refer to the section 3.2. Abstraction of the BERA: BERA Object Model) According to its development, the BERA Language will also be expanded to other building objects. In this study, the initial BERA Language and its tool implementation aim to handle a subset of building objects: space objects, group of spaces, their properties, and relations, in order to demonstrate the application for evaluating building circulation and spatial program rules. (Refer to the section 5.6. BERA Language Extensibility)

1.1.3. Domain, Scope and Primary Goal

The BERA Language design is intended to be open-ended to cover entire set of building objects, and attempts to demonstrate its implementations focusing essentially on the evaluation of building circulation and spatial programming. It is particularly appropriate for large facilities comprising a multitude of space instances and a large

number of requirements in terms of building circulation and spatial programming. The author and his team have been involved in the US Courthouse Design Guide Automation project funded by the US GSA [GSA-GT, 2010], and several related software have been implemented for actual use in new courthouse projects such as spatial program validation and circulation and security checking, etc. These two domains are plausible target applications for the initial BERA Language implementation.

The theoretical goal of this study is to provide a building object oriented approach to design a high-level and domain-specific computer rule language. It can be interpreted by general-purpose languages such as Java and compiled to be executed. The practical goal is to accomplish its effectiveness and ease of use without complicated knowledge of languages that are used in defining building information models and their manipulation. To achieve this goal, the example domain of BERA will be implemented and demonstrated, but the domain of BERA basically aims to be an entire set of building elements. In other words, the BERA Language has been designed focusing on building elements and its initial implementation described in this dissertation has specific focus on the spatial objects with their associated properties and their relationships derived from given BIM models, rather than two-dimensional footprints of the tangible building elements such as walls and slabs. Figure 1.3 describes the scope of the BERA Language design and the implementation in this dissertation, in the perspective of building objects, their properties and relations.

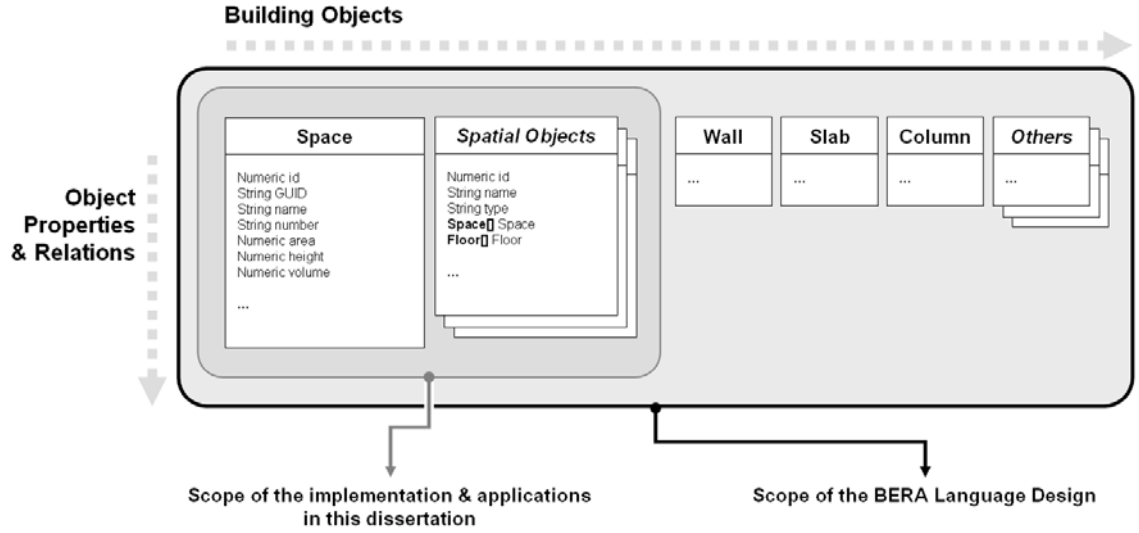


Figure 1.3. The building objects, their properties and relations within BERA. The BERA Language design aims to be open-ended. This dissertation will demonstrate its applications for evaluating the general domain of building circulation and spatial programming rules.

This study consists of two main parts: 1) description of the design and formal definition of the BERA Language, and 2) implementation of the BERA Language Tool. The former part attempts to answer the research question, which involves the effectiveness, ease of use and extensibility of the language for front-end users. This front-end part is standard for all other implementations. The latter part is a practical and technical guideline for the actual development of the implementation. Implementation issues are mostly related to the building information model scheme and its bridge to the language, portability of the language, platform-dependent issues, and so on. This latter back-end part varies by implementation environments. The target implementation of this study is based on IFC as given building information models, SMC as an IFC engine, and the Java Virtual Machine (JVM) [Java, 2010] as a compilation and execution environment. Considering various future environments of the BERA Language execution, there also can be another middle-level part: meta-information part for the BERA

Language. This part is defining the functions and operators within the language that determine the semantic scope of a version of the BERA Language. This defines what the back-end part should be. This is a BERA-specific meta-information part for the actual execution of the language in use.

Target users of the BERA Language are domain experts such as architects, designers, reviewers, owners, managers, students, etc., rather than BIM software developers. It means that the people who are interested in the building environment rule and analysis are the potential users. The rules implemented and applied in this study involve building circulation and spatial programming. This dissertation presents pragmatic applications that define rules for these topics and evaluate them using the BERA Language Tool (Refer to the section 5.5 and chapter 6). This tool comprises many libraries to alleviate common but unnecessary problems and limitations that are encountered when users attempt to analyze and evaluate building models using commercially available tools. Combined with other additional libraries which populate rich datasets for certain purposes, the BERA Language Tool will be fairly versatile to define rules and analyze various building environmental conditions.

1.1.4. Research Questions Regarding Language Design and Implementation

Designing a computer language and its implementation is not work that can be simply done by a couple of people in a short timeframe. A well-designed language might lead to a time consuming implementation process, but the main problem is that it is really difficult to design a “good” language. Fortunately there are several useful utilities, libraries and source codes that can provide assistance in the implementation languages under the agreement of open or limited licenses. However even the most powerful and

well-developed language utilities do not support language design per se. Moreover, constant updates on the language are required. Setting aside implementation issues, designing a language that is effective in specific domain problems and easy for novice users is a top priority. To accomplish this goal, and based on several reviews and projects done by us in recent years, the research questions can be summarized as follows:

- 1) In order to provide ease of use for BERA users, what aspects should be emphasized during the language design and implementation? Moreover, how can the factors in the BERA Language design and implementation be prioritized to achieve language effectiveness?
- 2) How can the level of abstraction be achieved and controlled when building a data model between computer-readable building information models (complex but explicit) and human-recognizable building data models (simple but implicit)?
- 3) How can a conceptual data model be conveniently and effectively accessed using BERA Language?
- 4) How are the primitive rules and functions defined so they can be easily composed into higher level design rules?
- 5) In order to implement the BERA Language efficiently, how can the execution pipeline be built as a road map for implementation?
- 6) How can the reusable language parser be implemented as the core language engine and portable applications, regarding not only BERA but also different BIM environments?

The scope of this study is the design and implementation of BERA Language. The scope of the implementation however cannot be usually clearly defined because of the nature of software development. Therefore this study will put emphasis on introducing a strategic way of one of the good implementations based on the implementation

environment. The capability of the BERA Language will be introduced and tested in the Application and Evaluation chapter, using a building model³.

Related materials of the BERA Language tool will be provided through the website under the partially limited license agreement (See Appendix D. BERA Language User Manual). The research questions will be resolved in the following chapters altogether.

Chapter 2 Background: reviews background studies and some significant examples for the BERA Language design, and attempts to answer the question 1). Chapter 2 is a survey to review related efforts in computer language design and implementation, and describes the lessons learned for designing and developing BERA Language. Specifically, it reviews how ECMA scripting languages are effective for handling target objects, and how Java-based domain-specific languages are translated and executed.

Chapter 3 BERA Language Design: introduces how to approach and accomplish good language design using fundamental design strategies. It also provides the core model for BERA: BERA Object Model (BOM), as an answer for the question 2), 3), 4), 5) and high-level issues on the lexical and syntactic design of the BERA Language.

Chapter 4 BERA Language Definition: provides the specification of BERA based on the context-free EBNF notation and associated texts. It also provides program examples in use and attempts to answer questions on implementation issues.

Chapter 5 BERA Language Implementation describes how the BERA Language design can be realized using associated utilities and libraries. It discusses some low-level

³ The language implementation is a huge area regarding its scoping, designing, and actual development. There are several useful and powerful tools to support developing this kind of domain-specific languages such as Lex/Yacc, Bison/Flex, JavaCC, ANTLR, etc, as a parser generator. The author took advantages of using a series of pre-set development environment for the research projects done by the team. See Appendix A and B for more information on the technical issues.

implementation issues regarding usability and expandability of the BERA Language. This chapter also describes the issues on BERA Language extensibility and portability for the further development. Especially, the section 5.6 attempts to answer for the question 6) regarding the extensibility and portability.

Chapter 6 Application and Evaluation demonstrates an actual building model and its applications. This chapter discusses some real-world rules and existing software-driven methods regarding the building circulation and spatial programming, and to demonstrate important capabilities of the BERA Language to handle rule conditions: how the BERA Language and its objects handle building objects, their properties, operations, and values. Accordingly, this chapter attempts to compare the language-driven method (the proposed BERA Language) with the software (parameter)-driven methods (SMC plug-in modules) that have been developed by the team including the author. It will demonstrate the capability of the BERA Language in terms of its expressiveness for addressing the complex state of a building and rules as the collection of rule conditions, using an actual building model example.

CHAPTER 2

BACKGROUND

2.1. Background

2.1.1. Overview

This chapter is a survey to review related efforts in computer language design and implementation, and describes the lessons learned for designing and developing BERA Language. According to the History of Programming Languages website [HOPL, 2010], its database lists 8,512 computer languages, with 17,837 bibliographic records featuring 11,064 extracts from those references. Wikipedia introduces over 600 languages under the page of computer languages, and it representatively lists 89 as commonly used computer programming languages to compare their general and technical information [Wikimedia, 2010]. Moreover, computer languages are born everyday [Mashey, 2004].

This study does not aim at tackling general and broad issues concerning programming languages. However, there are many useful references to study and develop a language as an extension of hundreds of languages that still have vitality. The BERA Language aims to be a domain-specific language dedicated to a particular field – the AEC industry sector. In order to make an influential effort, the BERA Language should reflect complicated and practical issues, but also be easy to use. This chapter describes the problems to be handled and the domain-specific modeling-based needs required to achieve this effectiveness and ease of use based on lessons learned from reviewing other types of languages.

2.1.2. The Problem

The main problem in domain-specific language development is usually how its effectiveness and ease of use can be accomplished. Programs should be written in high-level operations in terms of building design rules and analysis rather than in low-level operations with a higher level of granularity. There are two main parties involved in this program language use. One is the building design and evaluation experts such as the architect or engineer, and the other is the IFC expert whose expertise is in software implementation. As described in Figure 1.2, the BERA Language allows for providing more control and options to the people who are in charge of the design review process. If a high-level domain-specific language is as complicated as other general purpose programming languages, there will be no advantage from learning or using it. On the other hand, if it is too limited to handle specific problems, it will not be as valuable, even if it is easy to learn and use. This problem is very difficult but should be resolved at the language design level in order to properly figure out the best compromise between these two conflicting issues. Even the U.S. NASA (National Aeronautics and Space Administration) uses its own domain-specific languages for controlling space shuttles to improve reliability and reduce risks, cost, and development time [Parr, 2007].

2.2. Domain-specific Languages within BIM

2.2.1. IFC as a BIM Domain-specific Modeling Language

In the 1970s through 1980s, when there were only a few computer-aided design (CAD) systems, there were early attempts to exchange a set of geometric data between different systems without data loss. Interoperability between CAD systems was the main motivation for developing standard product models in each domain-specific sector. Since then, IFC has been developed by the International Alliance for Interoperability (IAI, currently known as buildingSMART) since 1994 for the AEC industry sector. IFC is one of the ISO STEP based modeling languages for representing building information models, and it is accepted as the dominant standard of building product model in the industry. The current version of IFC is 2x3TC1 as of 2010. By implication, it is not a closed definition. New versions will be released with industry-agreed and required features. Based on IFC models, a building model represents a lot of geometrical and topological information about the building as well as its associated properties. (Refer to the section 3.2.3 and Figure 3.1 as an overview how current IFC represents building objects regarding the scope of this study.) There are many materials on the IFC such as [ISO 10303, 1994; Eastman, 1999; IAI, 2000; IAI, 2003; Eastman, 2007; Eastman et al, 2008; buildingSMART, 2010a; buildingSMART, 2010b].

2.2.2. BIM Domain-specific Programming Languages

In contrast with the solid research and development efforts in the area of BIM domain-specific modeling languages, there is very little research and development in the area of BIM domain-specific programming languages. (Refer to Figure 1.1 for differentiating the BIM domain-specific modeling and programming languages) In 1970's, regarded as very early phase of the building information modeling research, there

was a significant effort on the BIM domain-specific language development. Eastman developed the GLIDE and GLIDE II (Graphical Language for Interactive DEsign) [Eastman and Henrion, 1977] as an attempt to generate building designs using a functional set of interactive operations. This effort was preceded by an earlier effort, called BDS (Building Description Systems) developed by Carnegie Mellon University [Eastman, 1975; Eastman et al, 1976] that was not a full language, but a set of operations and objects. GLIDE was well known and implemented at over eight other university research groups..

In the Geographical Information Systems (GIS) field however, efforts have been undertaken to develop languages and systems for manipulating geographical spatial data such as Spatial SQL [Egenhofer, 1987, 1994]. These systems contain spatial data such as the position and shapes of cities, streets, rivers, parcels, etc. They support only spatial objects in 2D because of the nature of this domain. These systems have been mostly developed based on relational database systems, and therefore their domain-specific languages have also been based on SQL language. For example, since the late 1980's, different dialects have been introduced such as Spatial SQL [Egenhofer, 1987, 1994], KGIS [Ingram et al, 1987], PSQL [Roussopoulos et al, 1988], GEOQL [Ooi et al, 1989], TIGRIS [Herring et al, 1988], and so on.

Not only in the field of GIS but also in the field of BIM, there have been significant attempts to design and implement a different type of “spatial query languages” for 3D building models, since building product model server environments have been used based on IFC, especially developed by: [Borrmann et al, 2009a; Borrmann et al, 2009b; Borrmann, 2010a] (See Figure 2.1). By using IFC, however, sometimes it is difficult to retrieve spatial semantics such as spatial topologies, relationships, and particular properties that are more important to domain experts. Examples are the Partial Model Query Language [Adachi, 2003] of the SECOM IFC Model Server [Adachi, 2010] and the Product Model Query Language of the EuroStep Model Server [Eurostep, 2010].

As the awareness of the importance of spatial semantics increased, some building model-centered queries were implemented based on SQL-like languages [Renz, 2002; Borrmann et al, 2006; Schultz et al, 2008]. Examples of spatial semantics based queries are as follows:

- Select all spaces in Level 2.
- Get all walls within the underground floors.
- Select all columns which touch this specific slab.

More specifically, an example in Figure 2.1 is a snapshot of the prototype application for 3D Spatial Query Language developed by [Borrmann et al, 2010a]. This is an excellent example to demonstrate how a language-driven approach to BIM is useful. The input program code used in this example is as follows. It retrieves all the column objects which are standing on top of a specific slab object.

```
SELECT Col.id
FROM IFCColumn col, IFCSlab slab3
WHERE ABOVE_HS_RELAXED(col.id, slab3.id) AND TOUCH(col.slab3.id)
      AND slab3.id = 'Oid23089_IfcSlab_Floor_'
```

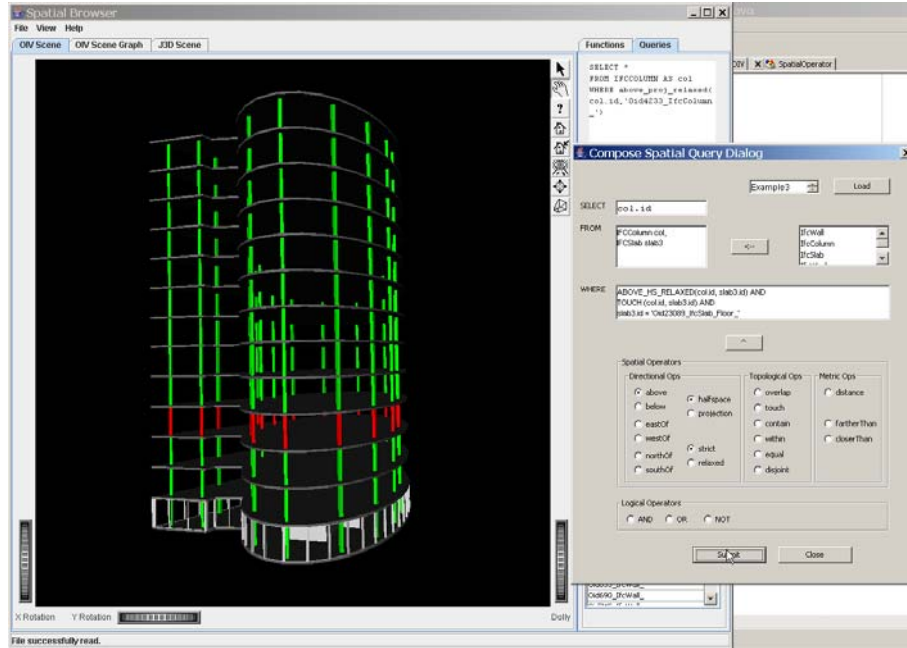



Figure 2.1. A screenshot of the 3D Spatial Query Language prototype application [Borrmann, 2010a].

This SQL-like query language on the building model is very easy and useful especially for the people who have experiences on databases and SQL query languages [SQL, 2010]. Also the prototype application shown in Figure 2.1 has a user-friendly dialogue interface to compose the language with ease. This kind of menu-driven interface is a useful utility for users even if its execution is eventually done by the form of program code. The BERA Language should have this query language feature within the language syntax and semantics because it is an essential feature for collecting user-defined objects⁴.

Recent efforts include the work of the Borrmann research group at the Technical University at Munich and VTT Finland to develop a “Spatial Constraint Language” for

⁴ Refer to the section 6.3 to preview how BERA Language queries and handles building objects. In the BERA Language, this querying feature is used for both object instantiation and rule checking. However, it does not mean that BERA Language is limited to the query language. The section 6.4, 6.5 and 6.6 demonstrate its rule definition and execution examples.

intelligent construction rule checking [Borrmann, 2010b]. This work focuses on spatial semantics and involves spatial query language based projects, but is still under development. Its scope is mostly on the building construction stage, and so building elements in this project are mainly structural objects rather than space objects. BERA on the other hand focuses on building environment rule and analysis, and the initial development is mostly on the spatial objects. Due to the nature of domain-specific languages, it is difficult to research and review a broad range of similar cases. However, based on the reviewed cases in this chapter, we could establish strategies, approaches and technical means to the BERA Language design and implementation.

2.3. Lessons Learned from Reviewing Other Languages

Although their boundaries are often too ambiguous to differentiate, computer programming languages can be classified based on several perspectives such as their chronology, category, generation, paradigm, target user, standardization, and so on. A standard language can even have many dialects for different purposes. This study introduces two sets of languages and represents what features can be borrowed to use in the design of the BERA Language. As a brief introduction, the first set contains JavaScript and ActionScript scripting languages in order to review their effectiveness and scalability according to the expansion of target objects. The second set is the Processing language that is one of the Java-based domain-specific languages in order to speculate how it is designed to alleviate difficulties for both users and developers. These languages are similar to each other in certain points because they are created to shorten the conventional language execution process such as edit – compile – run cycle [Scott, 2005].

Especially the scripting languages aim to support software applications quicker and easier by end-users.

2.3.1. Scripting Languages Based on ECMAScript

ECMAScript is a widely used scripting language especially for the World Wide Web. It is commonly used in well-known dialects of JavaScript and ActionScript. It has been standardized by ECMA International in the ECMA-262 specification and ISO/IEC 16262 in 1997 [ECMA International, 2010a; ECMA International, 2010b; ISO 16262, 2010]. Relying on World Wide Web Consortium (W3C) [W3C, 2010a] supported various Web standards and APIs, ECMAScript became one of the most popular scripting languages to users. JavaScript is supported in many common web browsers for handling client-side web documents. Most HTML [W3C, 2010d]-based web page developers are familiar with JavaScript and know how to handle web-document elements such as text field, image, button, selection box, and other various web-forms. JavaScript recently has been more and more popular because there are many existing standard-based new ways for developing web interfaces within the Web 2.0 [Web2.0, 2010], such as XHTML [W3C, 2010d; XHTML, 2010], CSS [W3C, 2010b], XML [W3C, 2010e], AJAX [W3C, 2010f], jQuery library [jQuery, 2010], and so on.

JavaScript was introduced in 1996 through the Netscape web-browser [Netscape Communications, 2010]. Due to the success of the World Wide Web, JavaScript has been popular to people even to those who are not familiar with computer languages. It is an object-oriented scripting language that runs on client-side web browsers to access objects within applications. It is influenced by many languages and its syntax is similar to the Java language as its name implies, and so it is easier for novice programmers to develop

dynamic web pages. It enables the control of elements in web pages such as W3C's document object model (DOM) [W3C, 2010c], and makes user interfaces more interactive. Figure 2.2 shows a simplified overview of DOM. It allows for accessing a page document and controlling its elements by means of scripts. JavaScript uses dot-notations to access not only the document's associated objects and properties but also the function calls in an intuitive way. This is also another important distinct feature of object-oriented concepts and implementation. An instantiated object has its properties as well as its pre-defined behavior (e.g. methods, function calls). Many recent object-oriented commercial programming languages commonly accept it simply because it is faster to write and clearer to read [Lethbridge, 2005]. For instance, based on the DOM in Figure 2.2, a navigation history can be accessed by the dot-notation: `window.history`, and a button in a form can be accessed by the notation: `window.document.forms.button`. The following JavaScript code shows an example of controlling an internal frame (HTML keyword: `iframe`)'s width and height dynamically, when the undetermined width and height of the internal HTML content is determined after page loading.

```
function reSize()  
{  
    var objBody      =    ifrm.document.body;  
    var objFrame      =    document.all["ifrm"];  
    objFrame.style.height = objBody.scrollHeight +  
        (objBody.offsetHeight - objBody.clientHeight)  
    objFrame.style.width  = '100%'  
}  
window.reSize();
```

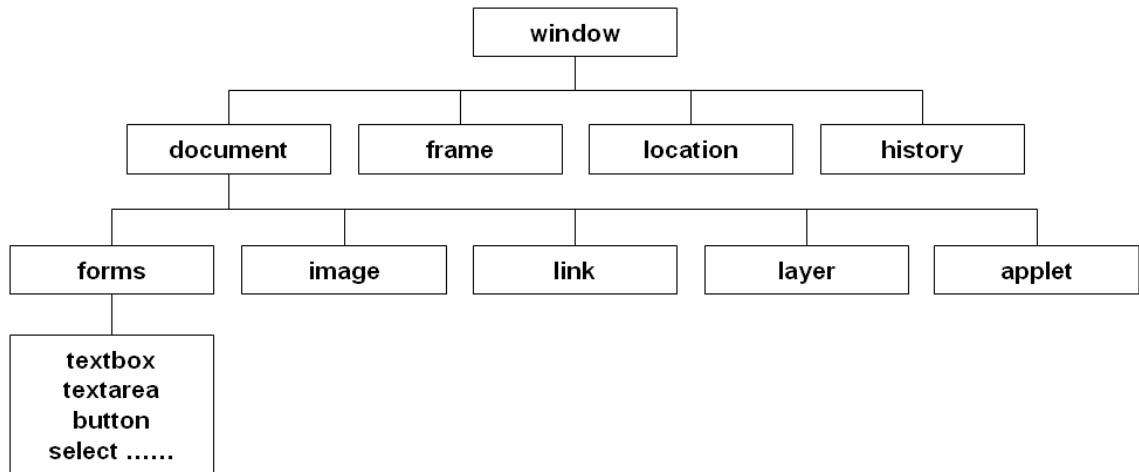


Figure 2.2. Example of the target objects of JavaScript: Document Object Model (DOM).

Another famous dialect of ECMAScripting language is ActionScript. It has also been broadly used mostly on the web, embedded in Flash movie clips [Gay, 2010]. As Adobe Flash™ (formerly Macromedia Flash) [ActionScript, 2010] has been used increasingly on the web, ActionScript became more popular in producing sophisticated movies. The following example shows a very simple action: if a user releases the mouse, the movie clip will be moved to the frame number 35 and starts playing. Idioms in ActionScript are very intuitive and user-friendly.

```

on(release)
{
    gotoAndPlay(35);
}
  
```

These example codes are very limited snippets of the two scripting languages. However, they substantiate their effectiveness and ease of use, which are the major problems that need to be overcome when developing domain-specific languages. As JavaScript controls web DOM, ActionScript controls the frames and behaviour of the movie. They are very effective and easy to handle specific problems in each target domain. As the nature of the scripting language and domain-specific language implies,

ECMAScript-based domain-specific languages influence many features for the development of the BERA Language, especially concerning their abstraction principle and means of handling target objects. BERA attempts to handle a pre-defined complicated model – the building information model. A building information model, such as one implemented in IFC, already provides a well-structured data format in its own scheme, but its internal data structure is usually complicated and heavy for casual users (See and compare Figure 3.1 with 3.2). In BIM software, building data structure is represented in an explicit way, but users tend to approach it in an implicit way because of the nature of a building design. ECMAScript stands as a precedent since it supports a pre-defined and standardized object model based on domain-centred user-defined names. The way target objects (DOM or Flash Movie) are managed by such scripting languages is not so difficult for domain experts. In JavaScript for example, users are accustomed to doing their job with target object models at a high-level. They do not need to comprehend how document models, client browsers, network protocols, and web servers are computationally inter-operated with each other at a low-level.

2.3.2. Domain-specific Languages Based on Java

A domain-specific language is usually hard to distinguish between small-sized general programming languages and scripting languages such as JavaScript. However, they are commonly found in modern computing environment. Examples include HTML embedded in web programming languages, CSS [W3C, 2010b] in web scripting languages, Regular Expressions [The Open Group, 1997] embedded in many other programming languages, and so on. Domain specific languages typically have the connotation of being smaller due to their specific purposes.

Due to the popularity of Java [TIOBE Soft, 2010] many domain-specific languages were developed based on the Java Virtual Machine (JVM), including Processing which is a Java-based DSL. Processing has been developed by Casey Reas and Ben Fry since 2001 at the MIT Media Lab [Processing, 2010]. Its target users are students, artists, designers, researchers, etc., who want to program images, animations, and interactive graphics. It originated from another domain specific language, Design By Numbers (DBN), which was developed by John Maeda in 1999 [MIT Media Lab, 2010]. Processing language is an easy language to learn for novice programmers, but creates powerful 2D and 3D graphics using few lines of programming. Because of its open-license policy and ease of use even for developers, it has a lot of dialects adopted by different domain users such as Wiring and Arduino for designing microcontrollers [Wiring, 2010; Arduino, 2010], and Fritzing to support physical prototyping for products [Fritzing, 2010], etc.

Processing has its own IDE (Integrated Development Environment) developed in Java, and it enables users to try it for their own interests. It executes source code and displays a graphical window using Java's Applet library that is also available through web browsers. Overall, the interface is fairly intuitive and easy. Processing program syntax is very similar to Java, and the codes are internally translated into pure Java codes and compiled by JVM for execution. This made Processing language very portable and manageable because it inherits benefits from the Java environment. Figure 2.3 illustrates an example of a Processing program that simply displays a rectangular and triangular shape on execution. The process shows the easiness of the language in both use and development, as most domain-specific languages aim to do. As shown in Figure 2.3, users only need to type simplified function calls such as `triangle(args)` and `rect(args)`, where `triangle` and `rect` are pre-defined methods in "PApplet" Java class file (they are also originated from Java 2D/AWT libraries), and `args` are their given parameters (in this case, all are 2D coordinate points or length values). The user input

code is translated into the pure Java code shown in (2) in Figure 2.3 and immediately executed by the JVM.

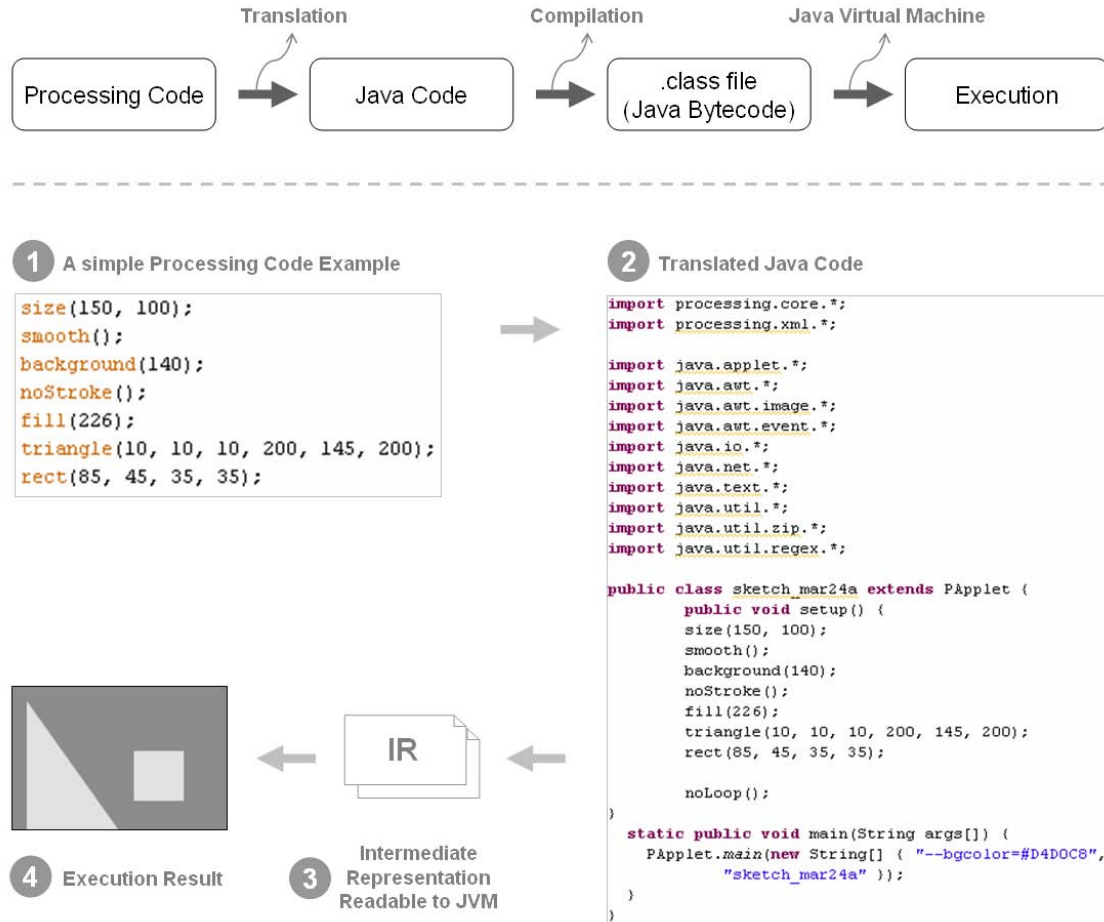


Figure 2.3. A simplified example of an execution of Processing language.

The architecture of this kind of Java-based domain-specific language has its influence on the development of the BERA Language in some ways, especially in actual implementation. However, the BERA Language has another huge layer of implementation: the building model platform and its bridges to the BERA Language. As reviewed in the section 2.2.1, IFC is a reasonable target building model because it is an effort to normalize the various native building models embedded in BIM platforms. Current BIM platforms have the translators that support IFC import and export, and IFC

is a close approximation to platform specific native building models. In the scope of this study, therefore the building model is in IFC format, and it basically transfers BIM data to be consumed in the framework of BERA Language. Accordingly, a simpler and neutral building model that is available to users is necessary. This model should be easily accessible to users instead of accessing either platform-dependent or usually very complicated native data structure including IFC. Moreover, it should still provide high fidelity to be useful on the specific problem domains even if it is simplified and neutralized. The next chapter describes this model (BERA Object Model: BOM) in detail. In order for it to be a portable and manageable language to different environments and implementations similar to the Java-based domain specific languages, the BERA Language implementation and application in this dissertation takes advantage of the JVM and its associated utilities.

CHAPTER 3

BERA LANGUAGE DESIGN

3.1. Design Strategy

3.1.1. Overview

What is good language design? There is no clear answer, but many researchers claim that there are important factors for a new language to gain acceptance and longevity. In terms of the main purpose of domain-specific languages, a new language should first effectively address new problems in an easy way [Mashey, 2005]. In this regard, BERA handles building models and aims to evaluate their validity and design performance. Second, a new language should substantially raise the level of abstraction for the complex state of the real world. As for the BERA Language, this denotes complexities of the building design. The BERA Language attempts to handle complicated building design issues that are mostly regarded as qualitative factors, in a well-transformed computational way. Moreover, a new language should be capable of handling new data types that are poorly or hardly addressed by current languages. BERA Language uses its own abstraction of the building objects as its basis data structure derived and computed from the given building model. (See 3.2. BERA Object Model) For example, BERA Language uses the “Space” object as a wrapped data type inherited from “BuildingObject” (in a generalized name in object-oriented concepts, a super class “BuildingObject” and its sub class “Space”) from the IfcSpace entity, similar to the way of other wrapper classes that are made up of an array or collection of user-defined data

types including primitive data types in recent general-purpose languages. Similarly, “Wall” object or any other type of building objects can be populated within the BERA objects, according to its domain rules and analyses. It is a composite data type which contains several properties required in building environment rule and analysis. The “Space” object is one of the most important data types in the BERA Language for evaluating building circulation and spatial programming rules in the implementation scope of this dissertation. A building is typically recognized in terms of semantics as a collection of spaces (rooms) during design by architects. It is usually spaces that are named or numbered for occupants (e.g. room names or room numbers) rather than bricks, walls or slabs, even if they are also named/numbered for building contractors. This building object-oriented approach affects fundamental issues in the language design: its objects, properties and operators. In order to design BERA as a domain-specific language, useful answers to this “good language design” question have been established in precedent case studies in the literature⁵.

3.1.2. BERA Language Design Strategy

The design and implementation of the BERA Language requires a fundamental design strategy the same as other type of languages. This section proposes a BERA Language design strategy by synthesizing the issues described in the introduction chapter and lessons learned from background survey chapter. As mentioned in the previous

⁵ The ultimate evaluation of the language and its design – the answer for a question: ‘Does BERA language have good design?’ – will come from the users. The open-ended testing, gaining feedback, and support arrangement are planned. See chapter 6 and Appendix D.

chapter, the main keywords in language design will be *effectiveness* and *ease of use*. According to them, before dealing with low-level strategy, high-level design strategy is summarized as follows.

- 1) Allow domain-experts who are novice programmers, such as architects or designers, to easily learn and write in the BERA Language.
- 2) Provide high-level methods to handle building objects and their properties with ease.
- 3) Enable the language to provide intuitive building and space object centered tokens/idioms rather than the vocabulary from building modeling languages, regarding the scope of the applications. E.g. an intuitive word “Space” instead of “IfcSpace” in IFC, “SSpace” in SMC, “Room” in Autodesk Revit, or “BSpace”.
- 4) Allow users to import and export external datasets for building space type definitions and/or pre-defined rules in the BERA Language definition. In these cases, a BERA user plays the role of a program executor, where he/she executes the rule checking based on pre-programmed rule statements on his/her own building models⁶.

The BERA Language is another way that users ‘talk to computers’ while using a given building model. That is why ease of use is an important keyword; however, it will be much better if BERA Language users have maximum control. Moreover, in terms of language extensibility, an open-ended and manageable BERA-specific data structure is required. This explains why the keyword effectiveness is also important. Taking into

⁶ As a preview of BERA Language, this explains the “BERA Reference” statement designed and implemented in current BERA language components (See the section 4.1 and 4.2). There are several building type-specific or domain-specific datasets need to be defined in advance for handling building type-dependent or domain-specific issues [J-K Lee, 2010b], and they can be handled by the “BERA Reference” statement with ease. The author realized this was pragmatically important in actual projects.

consideration all these aspects in the design stage, design strategies should be established. A low-level design strategy is described below. They will be addressed in each section in this dissertation, and the conclusion chapter will review them.

- 1) To define an open-ended object model for BERA Language. The domain mainly consists of building objects, group of building objects, objects relations, as well as their detailed properties. These should be expandable.
- 2) Operators on the BERA Object Model should be properly provided with ease of use. They are not only basic operators such as retrieving the single property of an object, but also complicated operations such as the dynamic instantiation of a group of objects, special extended structures such as circulation paths, adjacency and distance calculation, etc.
- 3) To define a “good” BERA Object Model, a certain level of abstraction for different building information models is consistently and generally defined. The data structure of IFC or native building models is usually managed differently in terms of low-level implementation, but BERA users should access them in a generic way. This is an important aspect for the portability and ease of use of the BERA Language.
- 4) The BERA Object Model should be capable of supporting many other types of building objects even if this study mainly focuses on spatial elements. This implies developing an open-ended and manageable BERA Object Model.
- 5) Higher level collections are necessary to handle BERA Object Models. They are dynamically instantiated at the BERA execution level, and are required to support additional operations. An example is the class ArrayList in Java [ArrayList, 2010], which is an ordered collection.
- 6) The BERA Language should support operational statements for advanced programs as other general languages do, such as logic operations, logic values, recursion, auto-iteration, auto-casting, negation, inheritance, polymorphism, and

so on, as well as basic algebraic operations. These should be fundamental constructs of the BERA Language in order to cope with arbitrary complex rules that can be defined and checked.

3.2. Abstraction of the BERA: BERA Object Model

3.2.1. Overview

One of the important factors to be considered in developing domain-specific languages is to provide high fidelity to problems. Not only is the capability of addressing new domain-specific problems necessary, but also the substantial ability to raise the level of abstraction on target elements is strongly required [Mashey, 2005; Parr, 2008].

The BERA Language has a strong bond to the building information model such as IFC. A given building model is always defined within its own data structure: native and partially-open data structures from native building models, open and neutral data structure from IFC, etc. This section describes how to build an abstract model from these existing building models, especially from IFC, named BERA Object Model (BOM). In the scope of this study and the initial implementation goal, we assume that given building models are IFC. Even if BERA users use only BOM, the IFC-oriented data structure still needs to be handled in terms of low-level implementation within the BERA as its back-end. Abstraction is an important process in language design, because some of the language keyword tokens and syntaxes will be derived from this model. In other words, this simple-yet-implicit abstraction will be used by front-end users, but the complicated-yet-explicit IFC data structure (or a host native object model) still needs to be accessed and used in the implementation stage of the BERA Language. The issues concerning the

resolution of these two different object models will be described in the following implementation chapter. The definition of BOM is open-ended, and the author realized that it is another challenge to define generic and valuable BOM as it grows more detailed. Within the scope of the implementation, this dissertation describes BOM structure focusing on spatial objects, as well as other building objects. (Refer to the section 5.6.2 Extensible BERA Object Model)

3.2.2. Building Environment Rules

Before handling building models, this section briefly addresses what are the “building environment rules”, and their relations with the building elements. BERA literally means analysis on building environment rules. As introduced in the former chapters, BERA Language attempts to define the rules and check them. In general, especially in the scope of this study, design rules are best practices, and all of the best practices are written in the form of design guide or regulations such as Hospital Design Guide [AIA, 1997], U.S. Courts Design Guide [USCDG, 1997, 2007]. They are the “rules” for design review. These best practices are usually building type-specific, and there is another type of rules within a specific binding code or regulation. Other rules are called codes. Codes contain the life safety requirements such as the Fire Code that regulates certain egress path-related conditions in case of fire [NY, 2004, OSHA, 2010], and the accessibility requirements such as the ADA related codes from Accessible Design Guide [US Access Board, 2002; 2006; 2010], etc. They vary from diverse kinds of requirements, and the author and his team have developed related design rules and requirements mostly focusing on building circulation and spatial programming issues. This section briefly examines how these rules can be represented in a computer-readable

structure using some example rules. Also, this issue is addressed in the section 6.2 in detail. More detailed issues are addressed in [Eastman et al, 2009b; Lee, J et al, 2010].

Here are some actual examples of the rules represented in natural language for the U.S. Courthouse designs.

- The size of a District Judge Courtroom is 2,400 NSF⁷. [USCDG, 2007 (Table 4.3)]
- The area requirement for the District Judge's Chambers Suite is total 6,000 USF⁸. (A specific requirement for the City of X Courthouse)
- The trial jury suites are accessed through restricted circulation corridors. [USCDG, 1997 (pp 3-14)]
- The judges' conference rooms must be accessible from judges' chambers suites by restricted circulation or a controlled reception area. [USCDG, 1997 (pp 6-11)]

The Georgia Tech team has developed automated building design review systems regarding the issues on building circulation and space program review for the U.S. Courthouses [Eastman et al, 2009b; GSA-GT, 2010]. Currently, the above rules are applied using two different programs, one for spatial area validation, the other for circulation and security checking [Eastman et al, 2009a; Eastman et al, 2010b; Lee J et al, 2010]. Because these rules are not computer-understandable statements, we have translated them in a certain structured form and put them into the parameterized table. The parameterized table defines what kind of building elements, objects, or properties are required for the rules, and it contains several different values for them. For example, above rules could be translated in a pseudo-program code as follows:

```
- Space("District Judge Courtroom"), NET_area = 2400
```

⁷ NET Square Feet.

⁸ Usable Square Feet.

- SpaceGroup("District Judge's Chambers Suite"), Usable_area = 6000
- Path("trial jury suite", "trial jury suite"), security = "restricted"
- (Path(Space("judges' conference room"), Space("judges' chambers suite"), security = "restricted")
OR (Path(Space("judges' conference room"), Space("judges' chambers suite")), mustHaveSpace = Space("controlled reception area")))

As shown in the examples, even if they are just snippets of the existing rules, rules basically deal with object properties that are possibly assigned on each space object or group of space objects. In these examples, objects are "Space" or "Path" which has two "Space" objects as its start and target spaces. The target object can be represented by given names with parenthesis, and its requirements and/or rules can be represented by a condition or set of conditions. At the bottom level after breaking down the rules into the atomic level, this "condition" is basically represented by an operation which consists of a left operand, right operand, their operator, and result⁹. The last example also shows various spatial properties, with a conjunction "OR" statement in its rule. To address the complexity of the rules, some capabilities should be seriously considered in the language design stage as follows:

- 1) How to provide easy access all of the object properties that the BERA Object Model defines,
 - 2) How to explicitly and effectively represent their requirements in the computer-readable and executable operations,
-

⁹ This is similar to the concept of ALU (Arithmetic Logic Unit) [Hwang, 2010] that is a digital circuit to perform arithmetic and logical operations, as one of the basic operations in computing.

- 3) How to allow rich predicates to express various kind of rule statements, e.g. conjunctions “AND”, “OR”, and the negation “NOT” for a series of operations,
- 4) How to deal with new properties that must be derived from others, and
- 5) How to generate new structures that can allow complex properties to be easily derived.

The third statement also includes logic operations, logic values, recursion, auto-iteration, auto-casting, negation, inheritance, polymorphism, and so forth, as adaptable features from recently developed and widely used languages. Especially, dot-notation supports any related access to the properties with quantification issues addressed in later sections. General construct statement definition in this chapter will define the second part. Such conditional statements and rich predicates will make BERA Language more expressive in its capability of handling building object properties and rules. Moreover, for being an extensible language, BERA Language also needs to handle additional objects and their properties that must be derived or computed from others. (Refer to the section 5.6 BERA Language Extensibility)

3.2.3. Building Model-centered Abstraction of the Building

As of 2010, the official latest version of IFC is 2X3 TC1. The official 2X4 version is under development for public release [buildingSMART, 2010]. The current version has a total of 653 entities that cover building objects, geometry, relations, and so on. According to its definition and hierarchical structure, a wall has a specific trace down the tree: IfcRoot → IfcObjectDefinition → IfcObject → IfcProduct → IfcElement → IfcBuildingElement → IfcWall. Figure 3.1 describes a simplified diagram for

representing the building elements within the IFC structure. It is an explicit and well-defined data structure for computers, not for humans. Many of the entities are omitted in the diagram. The highlighted objects are the subset of objects that require handling within the scope of the BERA Language.

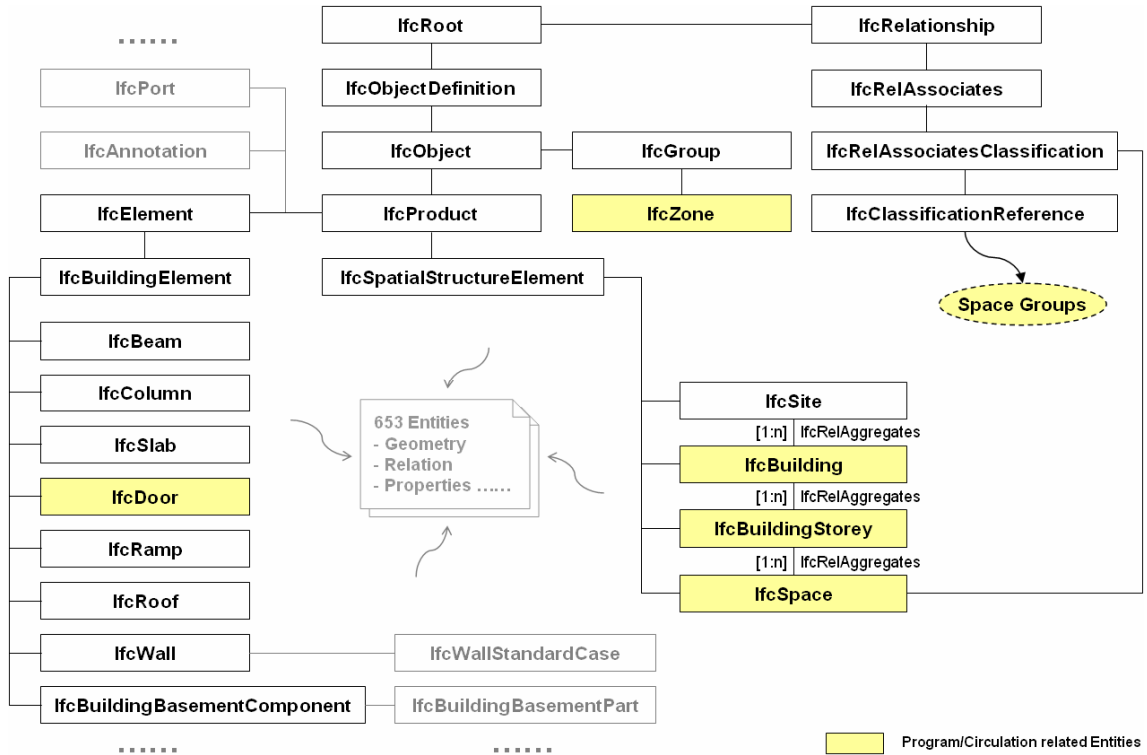


Figure 3.1. A Simplified IFC Entity Hierarchical Structure for the Building Elements.

3.2.4. BERA User-centered Abstraction of the Building

Target users of the BERA Language are domain experts who are interested in reviewing building models, rather than the computer programmers who are familiar with the IFC data structure such as Figure 3.1. They are architects, engineers, supervisors, managers or building owners. We propose an abstract model of the buildings for these

domain experts, as illustrated in Figure 3.2. It is comparatively much easier to comprehend, while the IFC structure in Figure 3.1 is hard to grasp in terms of its hierarchical structure and relation.

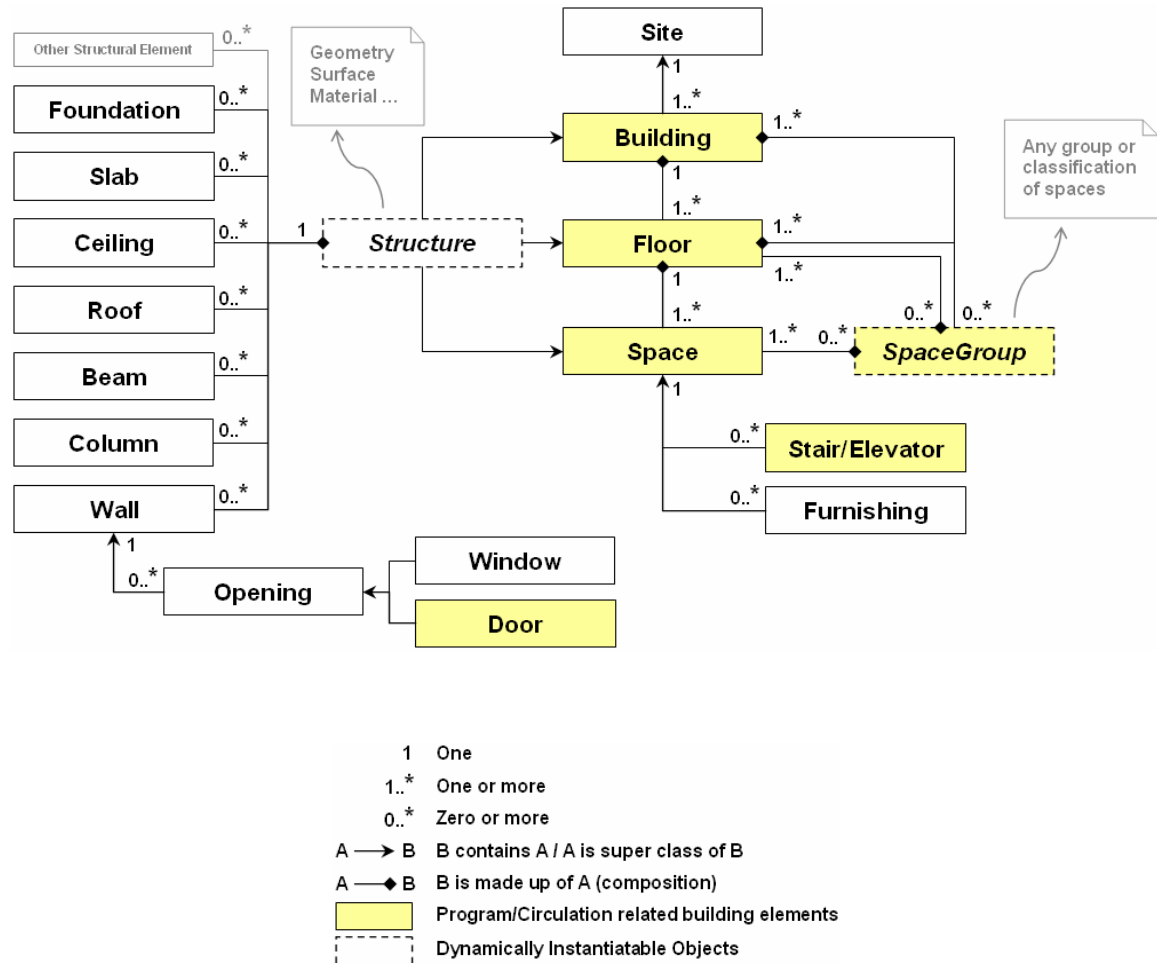


Figure 3.2. An Abstraction of the Building Model for BERA Users (BERA-centered).

A building contains diverse kinds of objects in multiple hierarchical structures. There have been many efforts in the area of building product modeling for the digital representation of buildings in various purposes [Eastman, 1999]. The main goal of this study is not to define the best building data model, but rather to achieve an intuitive abstraction of buildings for the usability of the BERA Language, similar to the Document

Object Model [W3C, 2010c] for JavaScript language. Figure 3.3 shows the subset scope of the BERA Object Model used within this initial study. A building has one or more floors, and a floor has one or more spaces. A space has zero or more stairs or elevators as circulation facilities, and it can have zero or more doors¹⁰. Stair and elevator objects are considered the determinants of the circulation type of spaces. Detailed properties of these objects will be described in the following chapters.

There is also an important object named “*SpaceGroup*”. Technically, it is a subclass of “*BuildingObject*” but dynamically instantiatable for representing any group or classification of building objects (in this case, they are all spaces). “*Floor*” is another case of *SpaceGroup*, but it is explicitly defined and statically determined when the building is modeled. As examples of domain-specific properties, “department”, “BOMA space category” [ANSI/BOMA, 1996] or “fire safety zone” can stand for a set of space objects for specific purposes. They are important because they determine other associated properties for each space for the purpose of space program review tasks. BERA defines a dynamically instantiatable object – *SpaceGroup*. In the IFC definition, as described in Figure 3.1, *ifcZone* is used for a similar purpose and the model view definition (MVD) attempts to make it a standard [MVD, 2010], but BERA regards it as belonging to *SpaceGroup* in this model. Any spatial group or classification can be instantiated in execution. BERA attempts to implement various type of space classifications based on these dynamically instantiatable objects. Similarly, with respect to building circulation, a path between two spaces can be defined in this dynamic classification using a series of space objects. Detailed implementation issues are described in the following chapters.

¹⁰ Theoretically a space must have at least one door, but the scope of this research is concerned with preliminary concept design where internal walls are not explicitly defined at this level of detail. Space objects are defined without internal walls, hence the condition of the zero-door space objects. Moreover, this relation between space and stair/elevator does not mean that stair/elevator is always subordinate to the space objects.

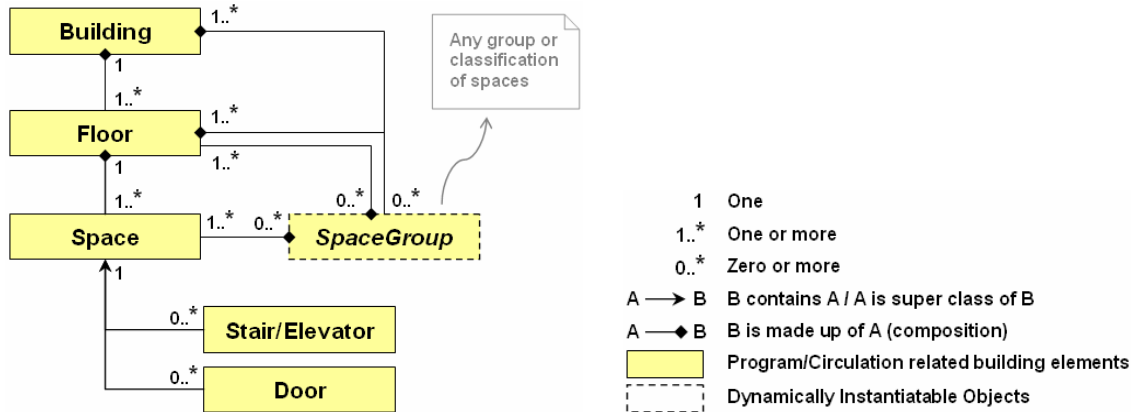


Figure 3.3. BERA Object Model design, within the scope of this study.

BERA users can take advantage of this BERA Object Model in the actual program. For example, if there is a space group named “department” and its name and associated properties are defined in IfcClassification, BERA users can retrieve the name of an instantiated space object space as follows:

- space.department

However, for the implementation of this code, BERA could translate it into an IFC-centered statement as follows (Refer to Figure 3.1):

- space.ifcRelAssociatesClassification.ifcClassificationReference.ifcLabel.getString()

In the implementation part, tracking IFC structure can be used in alternative forms, but this dot notation based example depicts how the BERA Object Model is intuitive and easy to use for BERA users. They only need to know the former statement. Details will be introduced in the following chapters and appendices.

Similar to SpaceGroup as a dynamically instantiatable object, the BERA Object Model offers another object – Structure¹¹. As shown in Figure 3.2, physical building elements can be grouped and instantiated through the “Structure” object in a dynamic way. This is primitively developed in this study, but it clearly claims that BERA can be expanded for different types of rules and analysis.

3.3. BERA Language Design

This section describes a lexical and syntactic design strategy for the BERA Language that addresses fundamental rules for naming and styling. As described in former sections, design strategy aims to obtain effectiveness and ease of use within the language semantics [Liu, 2003]. The BERA Language definition and implementation are described in following sections.

3.3.1. Lexical Design

This section describes the lexical design style for basic tokens and idioms that will be used in the BERA Language. Basically the flexibility is one of the keys in naming, but the fundamental strategy can be as follows:

- 1) The key here is simplified subject-verb-object (SVO) structures in general statements, as general computer languages conventionally follow.

¹¹ It can be “Assembly”, “Elements” or just “ObjectGroup” instead of “Structure”. Within the scope of the implementation, the lexical definition for the group of structural objects and dynamically instantiatable building elements are also open-ended.

- 2) The BERA Language assumes that S is the BERA Framework or the user; therefore, the VO structure will be used in a basic form of idioms in the BERA Language.
- 3) To reflect a universe of discourse in BERA, basic tokens in the BERA Object Model (BOM) will be the fundamental lexicon.
- 4) Object names will be starting with a capital alphabetic letters, and properties will start with a lower case letters, similar to the Java/C convention.

Lexical design and verbal tokens can be summarized as follows: (Refer to Figure 4.2 and 4.3 for actual definition and an actual implementation)

- 1) Building element objects as defined in the BOM definition:
Building, Floor, Space, SpaceGroup, Path, etc, as BOM defines.
- 2) To represent BOM properties and relations: dot-notation
- 3) To instantiate user-defined Rule: Rule
- 4) To acquire user-defined object: get (arguments)
- 5) VO structure also can be pre-defined as basic methods as follows: getSpace, getFloor, getPath, etc.

3.3.2. Syntactic Design

This section briefly previews a syntactic design style for the BERA Language as a preview of the language definition and implementation. The detailed syntax of the BERA grammars will be defined in the next chapter: BERA Language Definition. (The formal grammar definition is described in Appendix C: BERA Language Grammar.) The dot notation will be basically used for tracking objects and retrieving properties as in general

object-oriented languages (Java, JavaScript, C, etc.). Basic statement types in BERA are building objects and rules. Reflecting typical class, method, or a user-variable definition style, a user-defined object can be represented as follows:

```
ObjectType name(args) {  
    Statement;  
    ...  
}
```

- Where “ObjectType”, “name”, “args” and “Statement” are non-terminal tokens.

A BERA example code snippet can be represented as follows:

```
Space myOfficeSpaces(Space ss) {  
    ss.name = "office";  
    ss.Floor.height > 12;  
}
```

For example, a non-terminal token `ObjectType` can be “Space” or “Floor” as defined in the BOM lexicon, and `name` can be a user-defined variable name, and `Statement` will be dot-notation access to BOM objects and properties. Statements such as `ss.name = "office";` are logical conditions filtering what object instances are selected. The conditions will be blocked by curly brackets. For the rule definition, `ObjectType` will be “Rule”, and the given arguments `args` can be represented in the parenthesis. This is a basic form of the instantiation, but BERA Language syntax also defines some shorter forms for users: similar to the so-called syntactic sugar. For example, just a name string-based definition will be the most frequently used definition of the user-defined BOM as follows:

```
Space name = getSpace("space_name_strings");
```

Where “*name*” and “*space_name_strings*” are non-terminal tokens and “*getSpace*” is a syntactic sugar method to collect spatial objects by a name. In this case, a user-defined object can be instantiated by a single-line statement. An example can be represented as follows:

```
Space offices = getSpace("office");.
```

In this example, the operator “=” is used for the assignment operator, however in the former example, it is the equal operator. It is same to Java and its dialects (so-called C family languages). As mentioned in the former chapter, Java has been popular in the last decade, and people have developed several Java-like or Java-based dialects for their own purposes. The issues on syntactic design should be always open-ended for evolution sake. This study focuses on the effectiveness and easiness of language rather than a new style of grammar, therefore the BERA Language syntax will follow the existing style of general purpose languages such as the dialects of Java.

CHAPTER 4

BERA LANGUAGE DEFINITION

4.1. Overview

This chapter defines the syntax of BERA Language grammar. A grammar deals with the syntax of a language, and it is a set of rules where each rule expresses the sentence of the language. The specification of BERA is intended to formally define the syntax and semantics of the language and provide preliminary documentation of its use. Full definitions¹² and some implementation guidance are also provided in appendices.

As reviewed in the former background and language design chapter, the syntax of BERA Language is structured similarly to most programming languages, especially the most recently popular object-oriented language such as Java and ECMAScripting languages. BERA Language is generally object-oriented, procedural, and strongly typed. The user-defined elements such as building object groups and rules must be declared before they are used. This is especially important because of BERA's association with various building information models such as the IFC. Each declared language element in BERA has a strong association with corresponding entities or properties in the given building model. BERA is a language for defining and evaluating rules, and the rules are dealing with as many properties as the BERA Object Model (BOM) can provide.

¹² The definition in Appendix C is one of the plausible implementations of BERA Language as of 2010 fall, in the form of EBNF context-free grammar. It is growing. This chapter describes it using its fragments; therefore there are some differences between them. (For clarity, some definitions in this chapter are dropped from the definition in Appendix C.) The definition in Appendix C is an executable definition that is actually used in the BERA Language Tool development described in the following chapters. For the up-to-date version or other resources, refer to Appendix D: BERA Language User Manual.

Therefore, defining BERA objects are fundamentally important for both rule definition and its execution. Also it is why BERA Language has a concept of procedural programming that is used as a synonym for imperative programming. In other words, as BERA Language is running on top of a given building model, BOM definition is the first step, and then rule definition will be followed using the declared objects. Final step is of course the execution of the rules. They have data dependencies [Booch et al, 1998]. These object declaration or rule definitions could be derived from the external dataset such as pre-defined library and external BERA program file, thus, “import” functionality is also required at the beginning of the BERA program statement. A BERA program basically has four components as follows:

- 1) BERA reference directive
- 2) BERA Object Model definition and declaration
- 3) BERA rule definition
- 4) BERA execution statement

The basic procedure of the BERA Language components is defined above; however, there is another important feature of BERA Language: nesting. Apart from the language paradigm of BERA is mostly on the concept of object-orientation; these procedural components can be nested in other components for making successive calls. In other words, object declarations could be made in the rule definition or execution statements directly, e.g. within a curly bracket or parenthesis. This chapter will describe some pragmatic program segments and examples in each component definition.

In this chapter, for the formal definition of BERA Language, some typographical conventions will be used to facilitate communication of the language specification. In

addition to the normal text, we depict the syntactic structure of BERA in a fixed width font (here Courier New):

```
EBNF lexical and syntactic definition statements13
```

We also provide example segments of BERA Language code, to provide examples of the use of the language. For this we use of bold fixed width language,

Program segment

The formal definition described in this chapter is EBNF-based ANTLR rule [Parr, 2008]. The term ‘rule’ here means low-level lexical and syntactic definition in this formal representation. ANTLR is a language parser generator and it follows conventional lexical and syntactic rules and conventions for representing a language grammar inherited from Yacc [Johnson, 1979]. According to the Yacc’s convention, non-terminals starting with lower case are syntactic rules, and upper-case non-terminals are lexical definitions. (See Appendix A and B for more information: the metasyntax of EBNF and ANTLR used in this chapter, for expressing the syntax of BERA Language in a context-free grammar.)

Most of all, a non-terminal syntactic rule `bProgram` is the starting point of BERA Language definition. A lower case ‘b’ means BERA and the rest of alphabets are simplified tokens to represent each syntactic component. Other four non-terminals literally mean the above four components. The formal definitions in this chapter are partially introduced by each BERA component. Full definition is described in Appendix C: BERA Language Grammar.

¹³ See Appendix A. EBNF Notation as a Context-free Grammar Definition.

As a brief overview of the EBNF notation, EBNF grammar sub-rules are as follows. The elements x, y, and z represent a grammar fragment. (See Appendix A)

$x y z$:	match any alternative exactly once.
$x?$:	x is optional.
$(x y z)?$:	match nothing or any alternative.
x^*	:	match x zero or many times.
$(x y z)^*$:	match an alternative zero or many times.
x^+	:	match x one or many times.
$(x y z)^+$:	match alternative one or many times.

[1] Definition: bProgram

```

bProgram
    :      bBeraProgram
    |      BERABEGIN ';'
           bBeraProgram
           BERAEND ';'
    ;

bBeraProgram
    :      bReference? bBOMDef? bRuleDef? bExeStat?
    |      ';'
    ;

```

A BERA program is wrapped with BERABEGIN and BERAEND for separating it with embeddable other language programs such as Java or C#. It is dependent upon the target language to be translated and compiled for the final program execution level. If such begin-statement is omitted, BERA recognizer regards all input codes as pure BERA

Language to the end of lines. Or, if there is a target-language separator such as `java`; or `csharp`; it means following syntax is the syntax of target language. Also semi-colon (`;`) is used for terminating a single statement, if there are no other code block indicators such as curly bracket or parenthesis. Detailed lexicon and syntactic rules of BERA Language are fully described in appendices¹⁴.

The four main definitions defined under [1] **Definition: bBeraProgram** will be described in following sections: [2] **Definition: bReference**, [3] **Definition: bBOMDef**, [4] **Definition: bRuleDef**, and [5] **Definition: bExeStat**. All the definitions and example program segments addressed in this chapter are syntactically correct statements, verified by the BERA Language parser.

4.2. Reference Declaration

The reference declaration (`bReference`) has two sub-components: reference statement (`bRefStat`) and building type (`bBuildingTypeStat`) declaration. They can be used as various references in actual BERA program. They are optional and selectively used by users. The reference declarations mostly depend on back-end implementation, and fairly useful to import external dataset. The main purpose is to enhance the usability of BERA Language without additional programs or configurations. For example, a

¹⁴ As shown in above `bProgram` rule, the notation is basically EBNF as a context-free grammar for defining BERA language in this chapter. The fundamental lexer and parser for BERA are implemented by the ANTLR as a parser generator, thus all the statements in this chapter for defining BERA language are actual codes executable in ANTLR based on EBNF. Detailed implementation issues are described in the following chapter, and Appendices chapter also provide more information on the BNF, EBNF and ANTLR, as well as the full source code for BERA language grammar, including ANTLR-specific codes and comments.

building type-specific external dataset for establishing space object semantics [Lee J-K, 2010b] can be loaded by the user.

First, `bRefStat` can be used in:

- Importing additional libraries similar to the directive in general programming languages such as: **import** in Java, **using** in C#, and **include** in C/C++.
- Importing external definitions for building type-specific or project-dependent spatial data,
- Importing external pre-defined rules for a certain rule scheme, and so on.

Particularly the keyword `reference` supports external data links through the internet protocols, similar to the way importing an XML scheme definition in its application. `bProtocol` and `bURL` defines it.

Second, `bBuildingTypeStat` can be used in:

- Defining building type using either double-quoted strings or pre-defined building type terminal tokens in capital letters.

`bBuildingTypeStat` provides a reference for the set of space types, properties and rules within a spatial analysis program. `bRefStat` statement allows many instances, but `bBuildingTypeStat` allows only one statement.

[2] Definition: **bReference**

`bReference`

```
:      bBuildingTypeStat bRefStat*  
|      bRefStat+  
;
```

`bBuildingTypeStat`

```
:      BbuildingType
```



```

        ( BID | bStringQuot ) ';'
    ;
bRefStat
    :   Breference bURL ';'
    |   Breference BID ( '.' BID ) * ';'
    ;
bURL    :   ' ' bProtocol bURLadd ' '
    ;
bProtocol
    :   BProtocol
    ;
bURLadd
    :   BID ( BID | '/' | '.' | '%20' ) *
    ;

BID      :   BIDprefix ( BIDprefix | BIDDigit ) * ;
BIDprefix :   'a' .. 'z' | 'A' .. 'Z' | '_' ;
BIDDigit  :   '0' .. '9' ;

```

In these syntactic rules, a lexical rule `BID` defines the identifier tokens for BERA and will be broadly used in the definition of BERA. It is a basic lexical definition can be instantiated by any of the variable names, space names, and rule names within BERA programs. Only `BID` will be used as a terminal token in syntax definitions in this chapter without quotation marks for clarity. Detailed definition of `BID` and related terminals will be defined and implemented in the actual lexer implementation level.

The reference declaration part mostly depends on its back-end implementation as same as other languages' similar directives such as 'import' in Java. The more implementations and libraries are available; the more useful BERA reference will be

applicable to several purposes. That is why BERA reference declaration supports not only the access to the local paths, but also the network access to the web URL. Some examples of the valid BERA program segments within the rule `bReference` are as follows.

```
1) reference bera.gsa.Courthouse;  
2) buildingType "Courthouse";  
   reference "http://bim.arch.gatech.edu/bera/USCDG.bom";  
3) buildingType "OfficeBuilding";  
   reference bera.general.Office;  
   reference "http://bim.arch.gatech.edu/bera/Office.bera";
```

4.3. BERA Object Model Declaration

4.3.1. BERA Object Model

BERA Object Model (BOM) is the core data structure for the BERA Language definition and implementation. It is a standardized data structure for all different BERA implementations and environments; while a given building model data structure varies with its BIM engine or platform. For example, as depicted in Figure 3.1, IFC data structure consists of over 600 entities to represent a building model explicitly. Such an external data structure is also important for implementing the back-end BERA Language Tool in low-level development, because it is the raw data of BERA Object Model. For any other types of native building models, each implementation should be able to convert them into the BERA Object Model. BERA users are supposed to access as much data as building model provides and the BOM model can access, through the standardized

objects – the BERA Object Model. This section describes how BERA Object Model could be accessed by users, rather than how to establish it. The chapter for implementation will handle pragmatic issues on handling given building model data, running intermediate executors, and establishing BERA Object Models. Therefore, all syntactic definitions and program segments described here are the ways to define and access BERA Object Model in actual uses. It is an open-ended scheme for other type of building elements respond to the extensibility of BERA Language, as described in the language design chapter and the implementation chapter.

Within the scope of this study, spaces are the main objects of the current implementation of the BERA Object Model. This section describes static/dynamic spatial objects and their associated properties. In the abstract world of BERA Object Model, a building is made up of building objects. Especially spatial objects are very similar to human convention, which are composed of floors and spaces, and each floor has another set of spaces. Space objects are instantiated in actual implementation as many as actual space objects are defined in a given building model instance. Floor objects are a pre-defined convention of a certain type of space group in the same floor, and there are still much other type of space classifications and groups required to be instantiated. For example, “department”, “BOMA space category” or “fire safety zone” can stand for a set of space objects for specific purposes. In reality, they are important because they determine each space’s other associated properties for space program review tasks. BERA defines a dynamically instantiatable object – ObjectGroup. Especially for spatial objects, it is SpaceGroup. BERA attempts to implement various type of space classifications based on these dynamically instatiatable spatial group objects. Figure 4.1 describes these spatial relations in a simplified UML [Booch et al, 1998] notation. A dotted box is dynamic object and others are static objects. SpaceGroup objects are optional, but if any one of them is instantiated, it will be made up of at least one or more space objects. It can be implemented using a generic collection object such as ArrayList

in Java [ArrayList, 2010]. Using dot notation to these space objects, the space group objects can track its component Space and properties for establishing their own properties. As shown in Figure 3.2 and 3.3, some space associated objects such as door and stair objects are also involved in the definition of BERA Object Model. They are used in defining topological connections of spaces, metric circulation graphs, and so on. Other type of building elements are also available centered around the space objects, as a given building model definition supports these relations to the spaces.

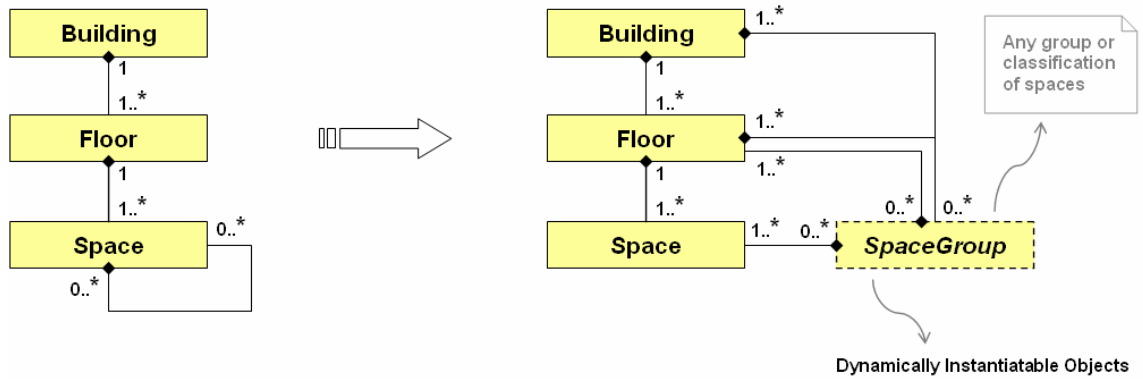


Figure 4.1. BERA Object Model within the scope of this study: spatial objects. A circulation path or a fire egress path is also another type of Space object, as a dynamically instantiated object.

4.3.2. Space and SpaceGroup Object

The main objects in BERA Language are basically “Building Object” and they can be grouped by various and dynamic assignments. In the definition of BERA Object Model, they called and dynamically instantiatable “Building Object Group”, and within the scope of this dissertation, they are SpaceGroup as described in Figure 4.1. The term instantiable means ‘able to be instantiated’ by dynamic calls in actual program execution,

as commonly found in the object-oriented programming [Gamma, 1995; Goodrich, 2005; Lethbridge, 2005]. Spaces are often referred to explicitly, and their proper naming in the building model is extremely important for a BERA program to operate correctly. In other cases, the spaces of interest are derived from the model, dynamically. An important example is the derivation of spaces used for circulation traversal from one space to another. In this case, there may be zero, one or many spaces (as defined within a specific building model) within the aggregated traversal space. In these cases, a traversal generates a space list (to be formally defined) that has an ordered list of spaces found in the traversal.

Following grammar definition depicts the declaration of dynamically instantiatable BOM, especially space objects within the scope. The second component `bBOMDef` in BERA can be defined as follows. Some non-terminal tokens are linked to the former definitions: e.g. the definition of `BID` is found in the definition [2]. The definitions in this chapter are a bit simplified definitions, but the full definition is described in the Appendix C that is executable by a general EBNF tool such as ANTLR. Some example program snippets are inserted within the lines starting with two slashes (that means ‘comments’). For example, following is an example of `bBOMDecLine`:

```
// e.g. Space officeSpace = getSpace("office");
```

[3] Definition: `bBOMDef`

```
bBOMDef
    :      ( bBOMDecLines | bBOMDefStat )+
    ;

bBOMDecLines
    :      bBOMDecLine
    ;

bBOMDecLine
```

```

// e.g. Space officeSpace = getSpace("office");
      :      bWrapSpaceType bDecSingle ';'
      |      bWrapStructureType bDecSingle ';'
      ;

bDecSingle
// e.g. allRooms = getSpace("room");
      :      BID '=' bBOMGetter
      ;

bBOMGetter
// e.g. getPath("office", "lobby");
      :      (bGetterVerbs bBOMGetterP ( ('+'|'-') bBOMGetter )?
      |      bBOMGetterExpr
      ;

bBOMGetterP
      :      '('
      ( bMultiBOMGetter | bBOMGetterExpr )
      ')'
      ;

bGetterVerbs
      :      bBOMgetBOM
      |      bVerbs ( bWrapSpaceType | bWrapStructureType )
      ;

bMultiBOMGetter
      :      bStringRep | bStringQuotRep
      ;

bBOMgetBOM
      :      bBOMgetSpace
      |      bBOMgetStructure
      ;

bBOMgetSpace
      :      BgetBuilding
      |      BgetFloor
      |      BgetSpace

```

```

        |      BgetSpaceGroup
        |      BgetPath
    ;

bBOMgetStructure
    :      BgetStructure
        |      BgetSlab
        |      BgetColumn
        |      BgetWall
        |      BgetDoor
        |      BgetStair
        |      BgetRamp
    ;

bBOMDefStat
    :      bDefineBOM
    ;

bDefineBOM
    :      bBOMDefStatDec bBOMDefBlock
    ;

bBOMDefStatDec
    :      Bdefine? bWrapSpaceType BID
    ;

bBOMDefBlock
    :      '{' ( bBOMDecLines | bBOMPropExpr )+ '}'
    ;

bBOMPropExpr
    :      bBOMGetterExpr ';'
    ;

bBOMGetterExpr
    :      bBOMGetterByBID
        |      bBOMGetterByProp
    ;

```

bBOMGetterByProp

//e.g. Space.Floor.name = "Level_1";

```
      :      bLogic?
          (bQuantifier '.')?
          ( bWrapSpaceType | bWrapStructureType )
          ( ('.' BID) | ('.' (bQuantifier '.')? ( bWrapSpaceType |
bWrapStructureType ) ) | ('.' BFunction) )+
          bComparisonOperator?
          (  BID    |   bStringQuot    |   '-'?   INTLITERAL    |   '-'?
DOUBLELITERAL )?
      ;
```

bBOMGetterByBID

//e.g. p.Space.area > 500.60;

```
      :      bLogic?
          (bQuantifier '.')?
          BID
          ( ('.' BID) | ('.' (bQuantifier '.')? ( bWrapSpaceType |
bWrapStructureType ) ) | ('.' BFunction) )+
          bComparisonOperator?
          (  BID    |   bStringQuot    |   '-'?   INTLITERAL    |   '-'?
DOUBLELITERAL )?
      ;
```

bBOMExpr

```
      :
          ((BQuantifier '.')? BID '.')?
          ((BQuantifier      '.')?      (bWrapSpaceType      '.'      |
bWrapStructureType '.'))+
          (BQuantifier '.')? (BID | BFunction)+
          ('.' BFunction)?
      ;
```



```

bQuantifier
    :      BQuantifier
    ;

bComparisonOperator
    :      ( '=' | '==' | '>' | '<' | '>=' | '<=' )
    |      bComparisonOperatorNegation
    ;

bComparisonOperatorNegation
    :      ( '!=' | '!==' )
    ;

```

In the actual BERA program, users can define spaces not only pre-defined explicit space objects directly from the building model space object, but also dynamically grouped space objects such as a circulation path, departments, etc. Some examples of the valid BERA program segments within the rule bBOMDef are as follows:

```

1) Space space1 = getSpace("office");
2) Space bigSpaces = getSpace(Spaces.area > 1000);
3) SpaceGroup commonSpaces = getSpace("corridor") +
    getSpace("lobby") - getSpace("private");
4) Path path1 = getPath("office", "lobby");
5) Space myOffice {
    Space.name = "office";
    Space.area < 500.0;
    Space.Floor.height > 10;
}
6) Floor lowGroundFloors {
    Floor.elevationHeight < 100;
    Floor.number < 6;
    Floor.number >= 1;
}

```

In the example 2), a user variable **bigSpaces** contains all the space objects that are bigger than 1,000 SF from the given model. This is an example of user-defined SpaceGroup object, as a collection of Space. The example in 4) is also SpaceGroup objects that have ordered Space objects – Path. Therefore in this case, BOM type declaration is Path. SpaceGroup object is handled by BOM handler in low-level with dynamically instantiated collection methods and auto-casting. Also this spatial object declaration supports arithmetic operations using plus or minus notation. It simply denotes that spaces can be added or extracted from a set of space groups to support user-defined dynamic spatial classifications. The actual program segments show these examples. Above 3) shows this case. The example in 5) describes a series of conditions for defining a SpaceGroup instance named “myOffice”, and the example in 6) is a user-defined Floor named “lowGroundFloors”. These 5) and 6) examples show that users can define their own dynamic and diverse space groups using as many as properties defined in the Space or Floor objects. All these spatial BOM have user-variable names, and will be used in the rule or analysis. More detailed aspects will be described in the implementation chapter.

4.3.3. Object Properties

This section describes BOM’s detailed properties as another important part of the BERA Object Model. Figure 4.2 and 4.3 describe detailed spatial objects within the definition of BOM. Fundamental properties from IFC are given data, and many additional domain-specific properties can be assigned on each object instance, as represented in boxes. For example, a space object can have GUID, name, number, area, height, and volume as generic properties. However a basic rentable area (RSF), as an example of a

domain-specific property, can be calculated by ANSI/BOMA related calculation method implemented in the GSA Extension libraries [ANSI/BOMA, 1996; Solibri, 2010; GSA, 2010a; GSA, 2010b]. Basically ifcSpace instance delivers a NET area, but basic RSF will provide additional area data for calculating usable or rentable area of the building. In this way, BERA Object Model has several more properties in each space element, grouped by four categories. First property type is a property set that is directly derived from the building model. Element's name, nominal area, related elements are in this category. Second property type is an additional property set that are acquired by additional calculation, computation, derivation from external dataset, etc. Examples contain usable square footage, space's assignable security type, BOMA category, and so on. Third one is the basic relational building element that is directly derived from the given building information. The last type is the additional relation that can be computed from given relations, such as the first floor of the building, adjacent spaces, directly accessible spaces, etc. Figure 4.2 and 4.3 show these examples on each BOM. In other words, the non-default property type could not be derived from the given model directly, but BERA libraries are in charge of computing them as an intermediate execution in the BERA Language Tool described in the following chapter (See BOM builder in the implementation section 5.3). More information on this type of properties is described in the implementation chapter and Appendix D: BERA Language User Manual.

Properties can be accessed via dot-notation approaches:

objectName.propertyName.

- where objectName and propertyName are non-terminal tokens.

Moreover, the dot-notation supports the link to the related objects, inherited object, function calls, etc., as other dot-notation based programming languages support. Examples regarding Figure 4.2 are as follow, where space and floor are the instantiated objects of Space and Floor respectively: (Refer to Table 5.1 for more actual use cases)

- 1) `space.height`
- 2) `space.security`
- 3) `space.basicRsf`
- 4) `space.Floor.name`
- 5) `floor.totalNetArea`
- 6) `floor.Space.name`

	Building	Floor	Space
Basic property	Numeric id String fileName Numeric <i>numberOf...</i> Numeric area Numeric totalNetArea Numeric height Numeric elevationHeight Numeric volume	Numeric id String GUID String name Numeric area Numeric totalNetArea Numeric height Numeric elevationHeight Numeric volume	Numeric id String GUID String name String number Numeric area Numeric height Numeric volume Numeric numberOfDoor
Relation	Site Site Floor[] Floor	Building Building Space[] Space	Building Building Floor Floor
Computed Relation	Floor firstFloor		Space[] adjacentSpace Space[] directlyAccessibleSpace
Computed & Derived property	String buildingType String designPhase String version Numeric buildingGrossArea Numeric structuredParkingArea Numeric mepArea Numeric skinArea Numeric externalWallArea Numeric biggestFloorArea ...	Numeric number ...	Numeric basicRsf String department String security String bomaCategory ...

Figure 4.2. Spatial objects and their properties: statically instantiatable BERA Object Model (BOM). (Refer to Table 4.1, 4.2 and 4.3 for detailed description for Building, Floor, and Space object)

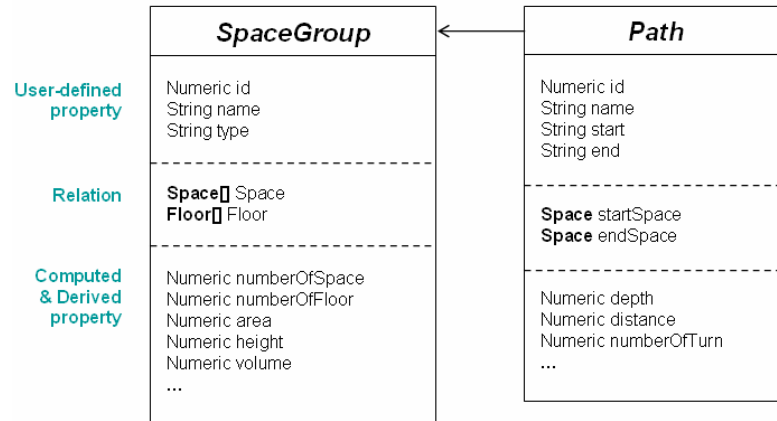


Figure 4.3. Spatial objects and their properties: dynamically instantiatable BERA Object Model (BOM). (Refer to Table 4.4 and 4.5 for detailed description for SpaceGroup and Path object)

The object classes SpaceGroup and Path in Figure 4.3 are dynamically instantiatable. Figure 4.3 shows their inheritance relation between SpaceGroup and Path, which means the all properties of SpaceGroup are available to Path. An instance of Path is a dynamically instantiated object by a user call of **getPath()**, similar to a SpaceGroup instance can be instantiated by **getSpace()** as defined in the definition above. A circulation path can be represented by a series of space objects as well as its start and end spaces, thus a path instance can be tracked back using its associated space objects. Examples regarding Figure 4.3 are as follows, where myOffice and myPath are the instantiated objects of SpaceGroup and Path respectively:

- 1) **myOffice.area**
- 2) **myOffice.numberOfSpace**
- 3) **myOffice.Space.name**
- 4) **myPath.distance**
- 5) **myPath.numberOfTurn**
- 6) **myPath.area**
- 7) **myPath.Floor.elevationHeight**

Figure 4.4 simply describes another link to the different type of object – Graph. A low-level graph based circulation analysis and related modules have been researched by the team in Georgia Tech. (Refer to the section 6.3.2 for the detailed description on the graph structure used in this BERA Language Tool development. Detailed issues are described in the papers: [Lee J-K, A, 2010; Lee J, 2010].) These kinds of meta-elements for the spatial representation or computation of additional properties are stored in BERA library, and will be called and used in actual BERA execution. They are currently building circulation-specific features of the implementation of BERA Language,

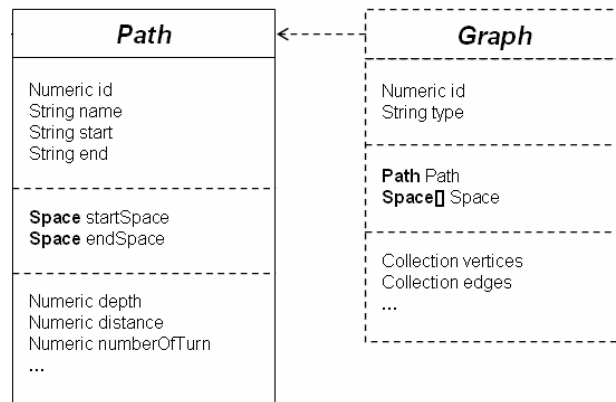


Figure 4.4. A circulation path, as one type of a dynamically instantiated SpaceGroup object, can be represented by a set of space objects, and it can be linked with other type of data structure such as Graph (dotted box) implemented by an additional BERA library for it.

The following Table 4.1 through 4.5 describes detailed descriptions on each property of the BOM: static BOM including Building, Floor, Space, and dynamic BOM including SpaceGroup and Path. The property name column contains the simplest dot-notation access to each property, e.g. Space.area. In actual use by the dot-notation expression, the dynamically instantiatable BOM objects such as SpaceGroup or Path can be replaced with user-defined BOM names: for example, “myOffices.area” or

“fireEgressPaths.distance”. The description column describes 1) its data type, 2) general description, optionally 3) example values for the property, and optionally 4) the BERA-specific methods to compute the values. The properties described in Table 4.1 through 4.5 are implemented in current BERA Language Tool, and of course they are open-ended data structure. Some properties are generic and some are domain-specific. The definition of these object properties is one of the main development works for the next version of BERA Language Tool. (Refer to the section 5.6 BERA Language Extensibility – lateral and vertical extensibility)

Table 4.1. Building properties and description.

Property Name	Description
Building.id	[Numeric] BERA object identification number starting from 1. The last id is always same to the total number of this BOM element. *In most cases, Building is one, but BOM defines them one or more as all other objects.
Building.fileName	[String] The name of this building model file.
Building.numberOf...	[Numeric] The number of object count. E.g. numberOfSpace, numberOfFloor, numberOfDoor, numberOfWall, numberOfStair, etc.
Building.area	[Numeric] The building gross area.
Building.totalNetArea	[Numeric] Total net area of the spaces in this building. The sum of the all spaces' net area.
Building.height	[Numeric] The height of the building from the ground. The sum of the all over-ground floors' height.
Building.elevationHeight	[Numeric] The construction height of the building including underground floors.
Building.volume	[Numeric] Total volume of the building
Building.Site	[Site] A site can have multiple buildings. (Similar to the Project. To be elaborated in next versions.)

Building.Floor	[Floor] Floor objects contained by this building.
Building.firstFloor	[Floor] Floor object that is the first level of this building. Computed by floors' elevation height.
Building.buildingType	[String] The type of this building. BERA building type declaration assigns this. Later, specific building type handlers will be updated in BERA Language Tool. E.g. "Office building".
Building.designPhase	[String] The design phase that is inferred by the object counts, relations, etc. E.g. "Late Concept Design".
Building.bimDesignTool	[String] The BIM authoring tool's name that exports this building IFC model. E.g. "Revit Architecture 2011"
Building.version	[String] The recorded version of the building model or its scheme. E.g. "IFC 2X3"
Building.buildingGrossArea	[Numeric] The building gross area. Currently this is same to Building.area.
Building.structuredParkingArea	[Numeric] Total parking spaces' area in this building.
Building.mepArea	[Numeric] Total mechanical, electrical, and plumbing spaces' area in this building.
Building.skinArea	[Numeric] Total building skin area. The sum of externalWallArea and topViewArea.
Building.externalWallArea	[Numeric] Total external walls' surface area.
Building.biggestFloorArea	[Numeric] The area of the biggest floor in this building.

Table 4.2. Floor properties and description.

Property Name	Description
Floor.id	[Numeric] BERA object identification number starting from 1. The last id is always same to the total number of this BOM element.

Floor.GUID	[String] Globally unique ID strings from the IFC.
Floor.name	[String] The name of this floor. E.g. “Level 1”
Floor.area	[Numeric] The gross area of this floor.
Floor.totalNetArea	[Numeric] Total net area of the spaces in this floor.
Floor.height	[Numeric] The height of this floor.
Floor.elevationHeight	[Numeric] The height from the ground to the bottom of this floor.
Floor.volume	[Numeric] The volume of this floor.
Floor.Building	[Building] Building object that contains this floor.
Floor.Space	[Space] Space objects that are contained by this floor.
Floor.number	[Numeric] A computed number assigned to this floor. The overground floors area always 1, 2, ...n, and underground floors are always -1, -2, ... -m. E.g. If this is the second floor, number is 2. If this is the third basement floor from the first floor, number is -3.

Table 4.3. Space properties and description.

Property Name	Description
Space.id	[Numeric] BERA object identification number starting from 1. The last id is always same to the total number of this BOM element.
Space.GUID	[String] Globally unique ID strings from the IFC.
Space.name	[String] The name of this space. E.g. “Office”
Space.area	[Numeric] The NET area of this space.

Space.height	[Numeric] The (ceiling) height of this space.
Space.volume	[Numeric] The volume of this space.
Space.numberOfDoor	[Numeric] The number of doors that are contained by this space object.
Space.Building	[Building] Building object that contains this space.
Space.Floor	[Floor] Floor object that contains this space.
Space.adjacentSpace	[Space] Space objects that are adjacent to this space.
Space.directlyAccessibleSpace	[Space] Space objects that are directly accessible to this space.
Space.basicRsf	[Numeric] Basic Rentable Area (RSF) of this space. See ANSI/BOMA standard. [ANSI/BOMA, 2010] An ANSI/BOMA specific property.
Space.department	[String] A department name assigned to this space. A BIM authoring tool-dependent property. E.g. “Office Area”, “Restricted Zone”.
Space.security	[String] A security level assigned to this space. A GSA’s circulation and security-specific property. E.g. “Public”, “Restricted”.
Space.bomaCategory	[String] ANSI/BOMA space category assigned to this space. An ANSI/BOMA specific property. E.g. “Office”, “Building Common”. See ANSI/BOMA standard. [ANSI/BOMA, 2010]

Table 4.4. SpaceGroup properties and description.

Property Name	Description
SpaceGroup.id	[Numeric] BERA object identification number starting from 1. The last id is always same to the total number of this BOM element.
SpaceGroup.name	[String] A user-defined name of this collection of spaces. E.g. “mySpaces”, “CirculationSpaces”.

SpaceGroup.type	[String] The type of the elements in this BOM. In this version of implementation, this allows only one type. E.g. “Space”, “Floor”, “Path”.
SpaceGroup.Space	[Space] Space objects that are contained by this collection.
SpaceGroup.Floor	[Floor] Floor objects that contained by this collection.
SpaceGroup.numberOfSpace	[Numeric] The number of Space objects in this collection.
SpaceGroup.numberOfFloor	[Numeric] The number of Floor objects in this collection.
SpaceGroup.area	[Numeric] The total area of this SpaceGroup. The sum of all spaces’ basicRsf.
SpaceGroup.height	[Numeric] The average height of all the spaces in this collection.
SpaceGroup.volume	[Numeric] The volume of this collection of spaces.

Table 4.5. Path properties and description.

Property Name	Description
Path.id	[Numeric] BERA object identification number starting from 1. The last id is always same to the total number of this BOM element.
Path.name	[String] A user-defined name of this collection of spaces. E.g. “myCirculationPath”, “fireExitPaths”.
Path.start	[String] A user-defined name of the start space for this circulation path.
Path.end	[String] A user-defined name of the end space for this circulation path.
Path.startSpace	[Space] The actually detected start Space object in this building.
Path.endSpace	[Space] The actually detected end Space object in this building.

Path.depth	[Numeric] The number of spaces between the start and end space.
Path.distance	[Numeric] The metric distance of this path instance. Algorithms are described in: [Lee J-K, 2010]
Path.numberOfTurn	[Numeric] The number of turns in this path instance.

4.3.4. Data Value and Operation

A dot-notation expression is the proposed easy and intuitive access to the BOM. But it is just a part of an operand that actually used in the BERA Language program for both defining dynamic BOM and rules. Especially, as shown in the definitions and examples in this chapter, dot-notation is the left-operand for describing a specific condition. See following examples in the form of dot-notation, operator, and value:

- 1) `Space.area > 500`
- 2) `Space.Floor.name = "Level 1"`
- 3) `officeSpaces.Space.name != "Lobby"`
- 4) `path.distance < 200`
- 5) `lobbyArea.height >= 10.00`

As dot-notation based access to BOM properties implies, the meaning of them can be intuitively read and written. As described in former sections, BERA Language offers simply two data types of property value: Numeric and String. For string expression, as same as other languages, double-quotation marks should be used to block given string values. For example, `Space.number` is String data type because space numbers are usually containing string characters such as N-101; therefore its data type is String rather than Numeric that is blocked by the double-quotation mark as follows:

```
Space.number = "302"
```

Left and right-operands are pairing each other with operators. Operators are data type-specific. For instance, angle brackets (>, <) should not be used for String data type. Operators can be used as follows:

Operators for String: =, !=, ==

Operators for Numeric: >, >=, =, <, <=, !=

These are as same as other common language operators. The notation “!=” means a negation (not equal) as same as Java syntax. Nothing is new in the operator notations in BERA Language syntax, but here is a BERA Language-specific note: equal and double equal notation has a BERA-specific and space name matching-specific function (In Java, double equal notations means “logical equal”). See Appendix D: BERA Language User Manual for this tweak. In actual uses, “name” strings are always problematic because they are human-read purposed unrestricted characters [Lee J-K, B, 2010], and the BERA Language syntax attempts to handle the problem using these equal notations. Reflecting space object semantics, BERA Language supports a functionality to handle the name string issue, therefore equal notation “=” is used as a meaning of “semantically same name”. For example, following operand will pick not only the spaces exactly named “office” but also “shared OFFICE-2” or “Tom’s Office”.

Space.name = "office"

However, for Numeric data type, equal notation has same meaning to others. If BERA Language users want to gather only exactly same string names, simply use double equal notation for String data type, especially for spatial names. This pre-defined functionality in BERA Language has been developed reflecting the author and his team’s lessons learned from the actual projects, and a subject to be updated for enhancing its capability of “grasping user’s intents when they type space name strings”.

4.3.5. Quantification and Conjunction for the BOM Expressions

This section describes some issues of the example rule application area focusing on the building circulation and spatial programming, but also available to other type of building objects or domain of problems. Basically people recognize a space not only a single room but also a group of spaces. In reality, people do not take consideration of computational space objects' quantity when they call a space, zone, or any of grouped spaces. For example, when people say a "floor common area", it usually means a shared area by different tenants on a floor, and computationally it is a collection of space objects including shared toilets, washrooms, closet, telephone room, mechanical space, elevator lobby, and public corridors on a same floor. If BERA strictly regulates the explicit quantification of space objects whenever they need to be instantiated in BERA Language, users should differently define them according to the return values' quantity. For example, if a building has 20 spaces which has the name "office", a BERA statement `getSpace("office")` will return 20 instantiated space objects in an array. Therefore, BERA Language uses always a collection of data for handling this kind of quantity issues especially for the space objects. This is similar to the concept of the wrapper classes in Java. Using this kind of wrapper enables people to alleviate suffering from various possible quantification issues such as null, zero, one, or many. Whatever the result values contain, they can be executed without errors. This is a simple yet important idea in BERA Language definition.

Space and any dynamic SpaceGroup objects will be instantiated using data collections, that is, possible quantity of a single instance can hold zero or many instances without any specific quantity declarations. In other words, people can type a simplified

statement using `Space` rather than using explicit declaration such as `Space[]` or `Collection<Space>`, as formally used in Java. This also enables some basic arithmetic operations on the space objects as described above. It will be used for manipulating dynamic space groups in certain purposes by adding or extracting some relevant spaces. Adding some set of spaces will establish another set of space group instances. For circulation and security checking, as another example, some spaces can be added or extracted for calculating a circulation path between two spaces in a special condition. As an example, see following example:

```
Floor.one.Space.name = "lobby"
```

This example is different from `Floor.Space.name = "lobby"`, which means basically all spaces' names should be "lobby" in a specific floor. Regarding the quantification issues, many examples will be described in the implementation and application chapter.

Another important concept for handling BOM in better expressive way is the OR conjunction. Here is a statement: "Offices should bigger than 300 and they should not be located in the underground floors". This can be represented by a following collection of dot-notation operands as described in this chapter, where "offices" is a dynamically instantiated `SpaceGroup` object as a pre-defined collection of office spaces:

```
offices.Space.area > 300  
offices.Floor.elevationHeight >= 0
```

Consider the changed statement from the above example: "Offices should be bigger than 300, or should be located in over ground floors". For this case, BERA Language supports OR conjunction as follows:

```
offices.Space.area > 300  
or offices.Floor.elevationHeight >= 0
```

In other words, AND conjunction is default, but OR conjunction should be written by users when it needed in this kind of conditions. Here is another statement with quantification issue: “At least one of offices should be bigger than 500, or should be located above the ground floor”. Let’s put this quantity in the dot-notation as follows:

```
offices.one.Space.area > 500
or offices.Floor.elevationHeight >= 0
```

4.4. Rule Definition Statement

The third component of BERA program is the definition of rules. The rule definition is similar to the BOM definition because it also takes advantages from the use of dot-notation access to BOM, as well as expressive notations by quantification and conjunction issues. It can be defined as follows, and some non-terminal syntactic rules are linked to the syntactic rules in the former definition: bBOMDef. The full definition of the BERA grammar is described in Appendix C.

[4] Definition: bRuleDef

bRuleDef

```
:      bRuleDefUnit +
;
```

bRuleDefUnit

```
:      Bdefine? bDefType (':' bRuleType)? BID '(' bParamDef
')' (EXTENDS BID)?
      '{' bRuleDefExpr* '}'
```



```

        ;
bRuleDefExpr
    :      bBOMPropExpr | bBOMDecLine
    ;
bParamDef
    :      bWrapSpaceType BID (',' bWrapSpaceType BID)*
    ;

```

To address complexity of the rules, as reviewed in the language design chapter, two main capabilities should be encoded in this definition: 1) to provide easy access to the objects' properties as many as BERA Object Model provides, and 2) to allow rich predicates to express various kind of rule statements, including logic operations, logic values, recursion, auto-iteration, auto-casting, negation, inheritance, polymorphism, etc. Dot-notation supports any related access to the properties with quantification issues addressed in former section. And general construct statement definition will define the second part. Such conditional statements and predicates made BERA rule definition expressive. (Formal definition of these constructs is omitted at this time in above EBNF, for focusing on four major components. This will be elaborated in the implementation.)

BERA allows inheritance of rules: BERA supports another instantiation using “extends” keyword. If another rule named `circRule2` needs to be defined using the user-defined rule `circRule1`, a statement `extends circRule1` can be used. Rule definition statements for `circRule2` needs to deal with only new aspects of rules on top of the rules from `circRule1`.

Some examples of the valid BERA program segments within the rule `bRuleDef` are as follows:

```

1) Rule myrule1(Space space1) {
    space1.area > 500;
}

2) Rule myrule2(Space space2) {
    space2.area > 1000;
    space2.Floor = "Level_1";
    space2.security = "public";
}

3) Rule circRule1(Space start, Space end) {
    path = getPath(start, end);
    path.Space.security = "restricted";
}

4) Rule circRule2(Space start, Space end) {
    path = getPath(start, end);
    path.Space.security = "restricted";
    path.one.Space.name = "gate";
    or path.distance < 10;
}

```

4.5. BERA Execution Statement

The last component of BERA program is the statements for actual execution of BERA: bExeStat. As same as the former three BERA components: bReference, bBOMDef, and bRuleDef, bExeStat is also optional, because the execution of program is not mandatory if the user defined rules for later uses as an example of ‘bReference’ BERA program type. The full definition of the BERA grammar is described in Appendix C. The definition of bExeStat is as follows:

[5] Definition: bExeStat

```

bExeStat
    :      (bExeStatUnit)*
    ;

bExeStatUnit
    :      bRuleExeLines
    |      bExeIfThenElseStat
    ;

bRuleExeLines
    :      bRuleExeLine ';'
    ;

bRuleExeLine
    :      ( bRuleVerb | BID ) '(' bBlockExpr (',' bBlockExpr)?
    ')'
    ;

bBlockExpr
    :      BID
    |      bWrapSpaceType
    |      bWrapStructureType
    |      bStringQuot
    |      bBOMGetter
    ;

bRuleVerb
    :      Bget | Bcheck
    ;

bExeIfThenElseStat
    :      bExeIfStat bExeThenStat bExeElseStat? ';'
    ;

```

```

bExeIfStat
    :      IF '(' bRuleExeLine ')'
    ;

bExeThenStat
    :      bRuleExeLine
    ;

bExeElseStat
    :      ELSE bRuleExeLine
    ;

```

Some examples of the valid BERA program segments within the rule `bExeStat` are as follows, where ‘mySpaces’, ‘officeSpaces’ and ‘myPaths’ are pre-defined BOM names, and ‘myRule’ and ‘myCirculationRule’ are user-defined rule names:

```

1) get(Space);
2) get(mySpaces);
   get(myPaths);
3) myRule("office");
   myRule(officeSpaces);
4) myRule(getSpace("toilet"));
5) myCirculationRule("lobby", "entry");
6) myCirculationRule(officeSpaces, mySpaces);
7) if myCirculationRule("courtroom", getSpace(Space.security =
   "secure")) else myCirculationRule("courtroom",
   getSpace(Space.security = "restricted"));

```

The example segments 4) and 7) describe the nested BERA Language components. A `bBOMDef` can be directly inserted as a parameter in a rule statement, and a dot-notation access to BOM also can be used in a certain parameter passing.

The BERA execution statement is also easy to read and write by novice users, but there is another way to handle and execute the BERA objects. It is the target language based execution statements as other domain-specific languages. For the advanced users, accessing BOM using target languages such as Java or C# will allow programming in lower-level. The BERA Language syntax definition is open-ended as other programming languages, but of course the target language syntax is depends on each language. The execution by target language is dependent not on the definition but on the implementation such as the BERA Language Tool. Following chapter will describe the BERA Language Tool as an implementation of the BERA Language definition, and the next application and evaluation chapter will show various example programs with their execution results.

CHAPTER 5

BERA LANGUAGE IMPLEMENTATION

5.1. Implementation Overview

The BERA Language has been designed and defined so that its implementation is portable to different building information modeling platforms. Figure 5.1 describes the high-level execution architecture of the BERA Language, and Figure 5.2 shows it in detail as an extension Figure 5.1. In the BERA Language implementation stage, two main environments should be covered: the platform-free environment (in other words, the BERA Language-specific and common front-end part for different platforms) and the BIM platform-dependent environment. The front-end part is standard for all other implementations, while the back-end part varies by BIM platform. The basis of the target implementation of this study is the Industry Foundation Classes (IFC) as given building information models, Solibri Model Checker® (SMC) as an IFC engine, and the Java Virtual Machine (JVM) as a compilation and execution environment.

Regarding implementation, the BERA Language architecture consists of two high-level components as described below.

- Front-end: BERA engine: This contains user-generated language programs, the BERA translator/interpreter to the target language, and other intermediate representations and executors for generating the BERA Object Model. This front-end engine is standard for all implementations and environments.
- Back-end: Custom engine: The BERA Language could not be executed without a given building model. The building information modeling engine is another huge platform. SMC is a candidate platform that the BERA engine uses, both in terms

of object models and applications. This back-end implementation varies by platforms, but the target intermediate model is always the BERA Object Model.

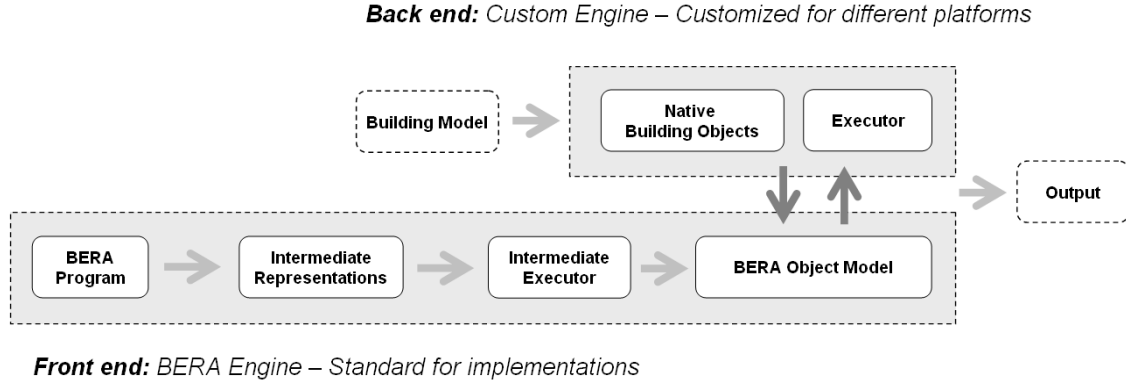


Figure 5.1. High-level Execution Architecture of the BERA Language. As a high-level data flow diagram, arrows mean data flow, and up/down arrows also include interactions.

This chapter describes one of the BERA Language implementation approaches regarding a plausible structure for the general language back-end issues: lexical analysis, syntactic parsing, semantic analysis, intermediate code generation, target code generation, and execution [Scott, 2005]. The back-end side issues and their implementation can be flexibly adopted by developers. The implementation approach described in this chapter is based on the actual application named the BERA Language Tool version 1.0¹⁵. This chapter focuses on generic issues of the BERA Language Tool rather than implementation and platform-specific details. Several example programs and their execution results will be described in the next chapter – Applications and Evaluations.

¹⁵ The current version of the BERA Language and its Tool described in this dissertation is the initial release as of 2010 fall. For the up-to-date version of BERA Language and its Tool, refer to Appendix D: BERA Language User Manual and its on-line resource website.

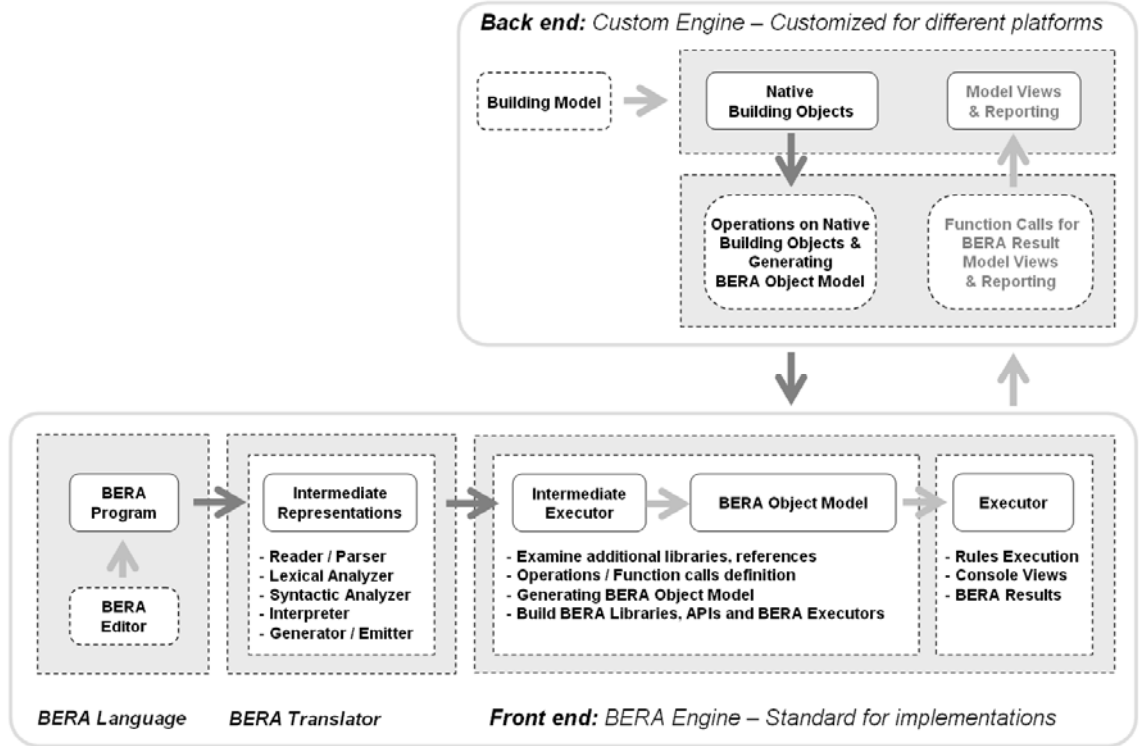


Figure 5.2. BERA Language Architecture Detail: Front-end and Back-end.

This chapter organizes and proposes some important features for describing the BERA Language implementation issues. The actual processes of the language implementation are combined in multiple iterated functions and performed simultaneously, but for clarity purposes they will be explained in this chapter as separate in sequence.

- 1) **BERA Listener:** BERA listener is in charge of lexical and syntactic analysis on the input program regarded as a language parser. This section has more focus on the semantic analysis and intermediate representations which translate the meaning of input texts into the executable BERA data structure: dynamic BOM and user-defined rule instances that are ready to be executed.
- 2) **BOM Handler:** As described in the Chapter Four, there are two types of BOM and their properties – static and dynamic. The static BOM handler computes all associated BOM data from the given building model, while the dynamic BOM

handler is in charge of collecting objects according to the user inquiries, using both static and dynamic BOM. There is another important handler – Rule Builder. It parses and builds the code for user-defined rules.

- 3) BERA Executor: The executor processes user execution statements as defined in the BERA Language syntax. The BERA listener and BOM handler already instantiate all relevant user-defined objects, and therefore the BERA executor is just a consumer in terms of the BERA Language execution.
- 4) BERA Language Tool: The BERA Language Tool is an integrated development environment for the BERA Language, and functions as an application with features described in this chapter.

This chapter emphasizes the BERA-specific front-end implementation issues in a platform-independent way, and address further implementation on different platforms and environments. This chapter also tackles some directions for the BERA Language extension. For details on the BERA Language grammar definition or use cases, see the next chapter and Appendices, especially Appendix D: BERA Language User Manual.

5.2. BERA Listener

5.2.1. Lexical and Syntactic Analysis

User textual input stream should be parsed and recognized before establishing its semantics and execution. The module named BERA listener is in charge of this first process of language recognition – lexical and syntactic analysis. Language recognition is

a very important step in any language application. Although this technical parsing has an important role in language implementation, it is beyond the scope of this work. This dissertation does not attempt to tackle general and detailed issues concerning parsing or its patterns. Some of the parsing modules in the BERA Language Tool have been facilitated by ANTLR [Parr, 2008; 2009; 2010] as a parser generator for user BERA Language input, especially for the lexical and syntactic analysis steps – the first stage of the BERA Language recognition. Instead, the author briefly introduces how these input texts can be handled in terms of building BERA semantics. The series of input tokens will be classified by the BERA listener as shown in Figure 5.3.

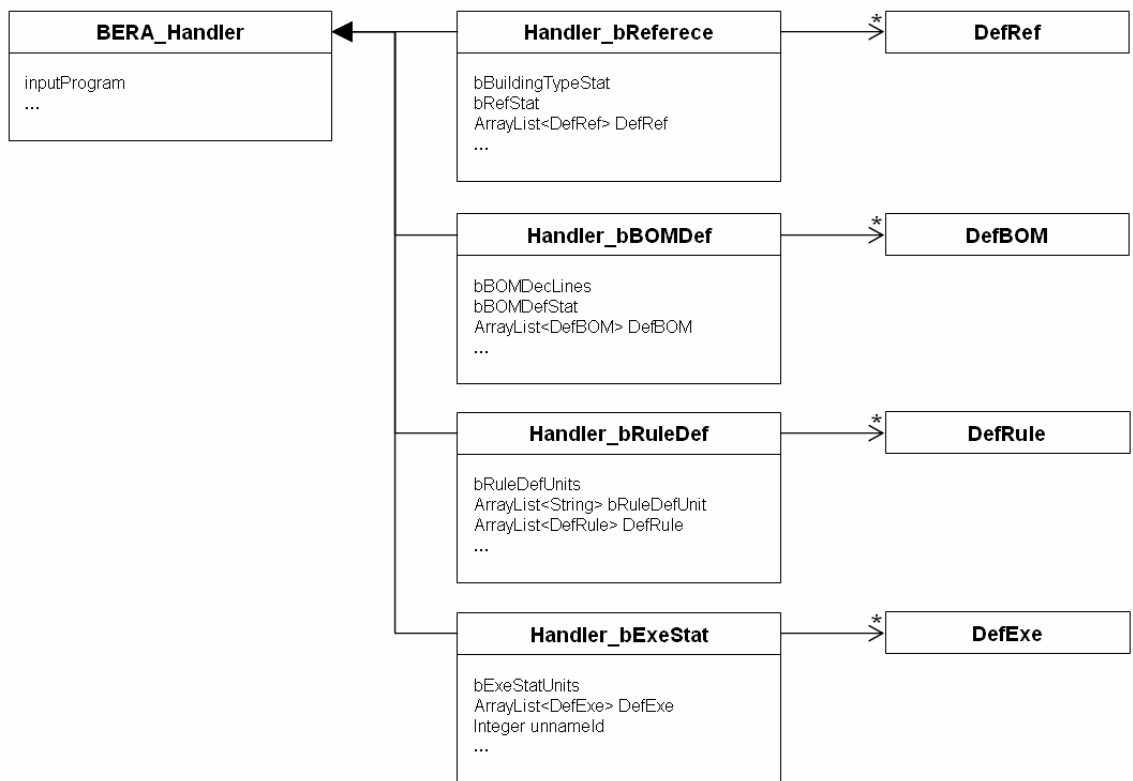


Figure 5.3. An overview diagram to describe the top-level lexical and syntactic analysis of BERA Language input program.

Figure 5.3 shows four components that are in charge of handling input texts. They are equivalent to the highest classification of the BERA programs as defined in section 4.1: bReference – DefRef, bBOMDef – DefBOM, bRuleDef – DefRule, and bExeStat - DefExe. Input texts are still considered unknown textual stream to computers. However, by using the BERA listener, they can be classified according to the high-level BERA Language structure as introduced in the language definition chapter:

- 1) DefRef (Reference Declaration),
- 2) DefBOM (Dynamic BOM Declaration),
- 3) DefRule (Rule Definition Statement), and
- 4) DefExe (Execution Statement).

Lexical and syntactic analyzers in the BERA listener play important roles to validate, report, and classify the user-input texts for the subsequent and more important task: semantic analysis.

5.2.2. Semantic Analysis

Semantic analysis is usually regarded as a final stage of language parsing. The BERA listener converts classified input streams into a specific data structure type such as the example shown in Figure 5.4. It describes an example of the actual data structure that has been used in the BERA Language Tool implementation. The objects described in the Figure 5.4 are all instantiatable objects to be triggered by each other, or consumed by the language execution statements.

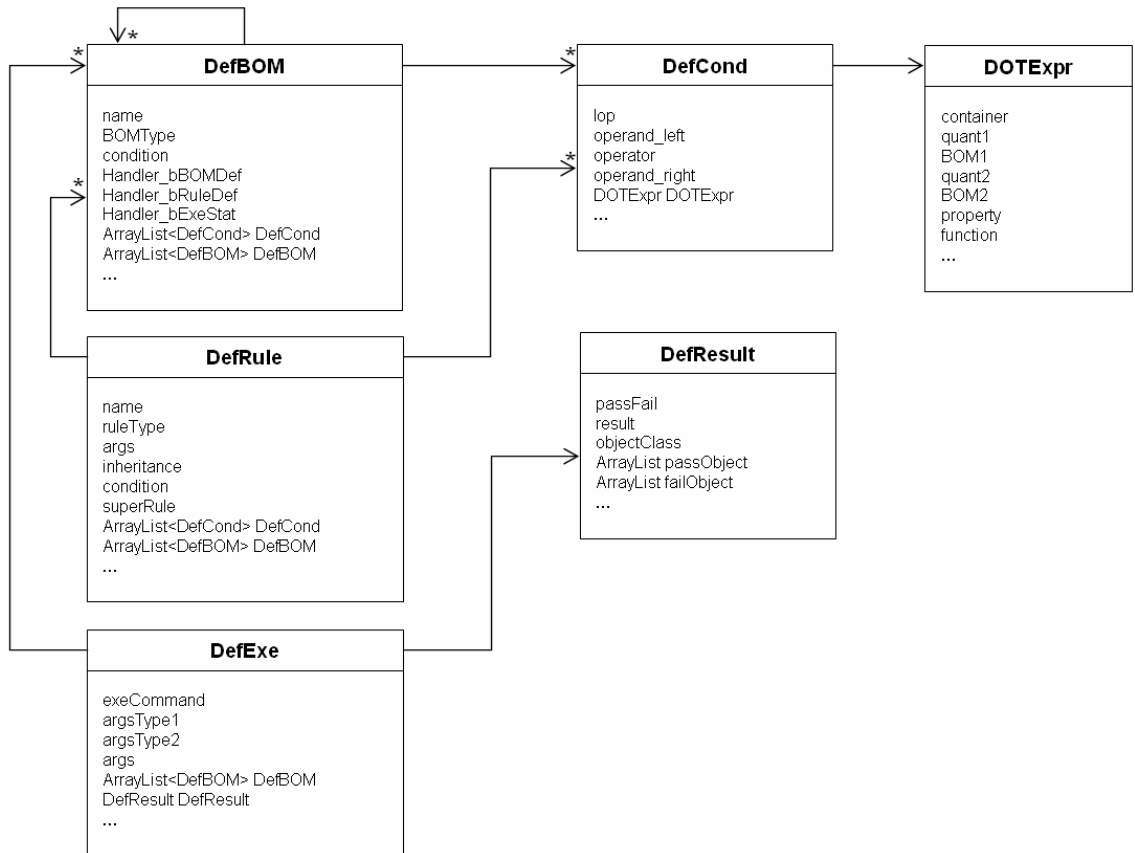


Figure 5.4. An overview diagram to describe establishing the BERA Language semantics.

As Figure 5.4 describes their hierarchy¹⁶, DefBOM, DefRule, and DefExe can populate multiple DefBOM, and a DefBOM and a DefRule include DefCond which is the condition definition statements. A DefCond contains a single operand for a certain condition using a dot-notation access to BOM, operator, and value. An important aspect of this DefCond is that it also has a one-to-one relation to DOTExpr which defines its semantic meaning of the dot-notation access to BOM. The execution statement DefExe has a DefResult to instantiate its execution result. The BERA listener works out the implications of these input components that are validated and takes the proper executions.

¹⁶ The component DefRef in Figure 5.3 is dropped in this diagram because it is a referential object. Its definition can affect the data structure shown in Figure 5.4, but still a subset of it.

For example, a single DefCond can be derived from a DefBOM or a DefRule. The following is an example of a DefCond:

```
path.one.distance <= 100
```

This example DefCond can be derived from a DefBOM as one of the conditional statements to define an instance of SpaceGroup, or from a DefRule as one of the rule conditions defined as a rule. The DOTExpr instance populated from this DefCond can be represented as follows (see Figure 5.6 for the structure of DOTExpr):

```
(container.)((quant1.)BOM1.)((quant2.)BOM2.)(property)
```

- Where all are non-terminal and optional tokens: For example, “path.one.Space.area” can be matched as follows: container – path, quant1 – one, BOM1 – Space, and property – area. (See Table 5.1 for more examples)

The semantic analyzer in the BERA listener is in charge of converting the text inputs that are processed by the lexical & syntactic analyzer into the object instances to be consumed in the language execution. This also contains most of the features of the semantic analysis such as BOM data type checking, the assignment of names or variables, and object binding. Figure 5.5 describes the DOTExpr parser which has the most important role in establishing BOM semantics. The semantic analysis can be done by early syntactic level or late intermediate representation/execution level, considering the implementation environment. The use cases and examples are introduced in the next chapter and Appendix D: BERA Language User Manual.

Table 5.1. Some valid examples of the dot-notation access to BOM (examples of DOTExpr in Figure 5.4). In the example expression column, words starting with upper case letters are BOM names, and words starting with lower cases are property names or quantification words. User-defined variable names for dynamic BOM are in *italic*.

Tokens	DOTExpr Tokens	Example Expression
1	container	Space
	container	<i>myOffices</i>
2	BOM1.property	Space.area
	container.property	<i>myPaths.distance</i>
3	BOM1.BOM2.property	Space.Floor.name
	container.BOM1.property	<i>myOffices.Space.height</i>
4	container.BOM1.BOM2.property	<i>myOffice.Space.Floor.height</i>
	container.quant1.BOM1.property	<i>myPath.one.Space.name</i>
5	container.quant1.BOM1.BOM2.property	<i>path.one.Space.Floor.height</i>
	container.BOM1.quant2.BOM2.property	<i>path.Space.one.Floor.height</i>

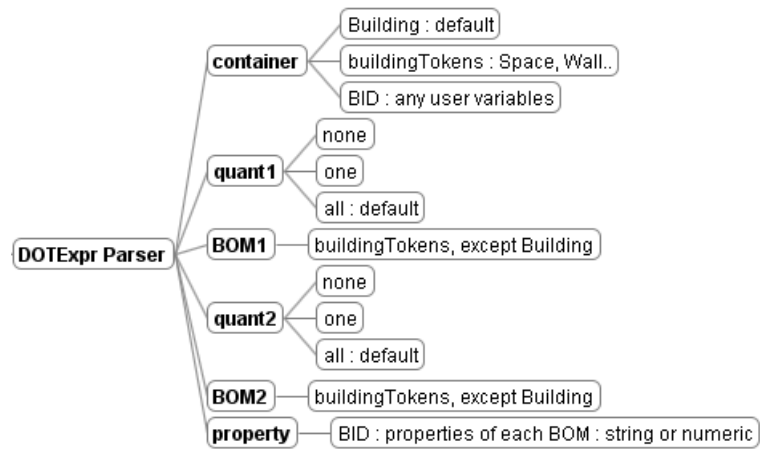


Figure 5.5. Overview of the structure of DOTExpr.

5.3. BERA Object Model Handler

The BERA Language is based on two types of BERA Object Model (BOM) – static and dynamic. Static BOM is a static data set from a specific given building model, and can be represented by class names such as Building, Floor, and Space as discussed in the language design chapter. The Static BOM is mostly pre-determined by the given building model, and therefore most of the given property values are statically established when the building model is loaded into memory. Some properties can be both statically and dynamically assigned by user inputs for further development. For example, “buildingType” under Building object can be assigned by users using Handler_bReference (See Figure 5.3), and “security” under Space object can be assigned by additional BERA library which is in charge of automated assignment of security level, even if their default values are empty. This is an example of the technical BERA Language extensibility for further use cases.

BOM builders are in charge of handling building objects and their properties. The static BOM builder is building model-specific; therefore there will be more emphasis in this chapter on describing the dynamic BOM builder. The focus of this study and implementation is on the spatial BOM such as SpaceGroup and Path as described in Figure 4.1, but the structural BOM instances (e.g. a sub class of ObjectGroup) are also instantiated and used in the tool because they have physical relations with spatial BOM instances. Figure 5.6 illustrates the BOM classes in the implementation described in this chapter. Briefly, the arrows between SpaceGroup, Path and others mean inheritance (the same as between Structure and others), and other arrows mean their association relation.

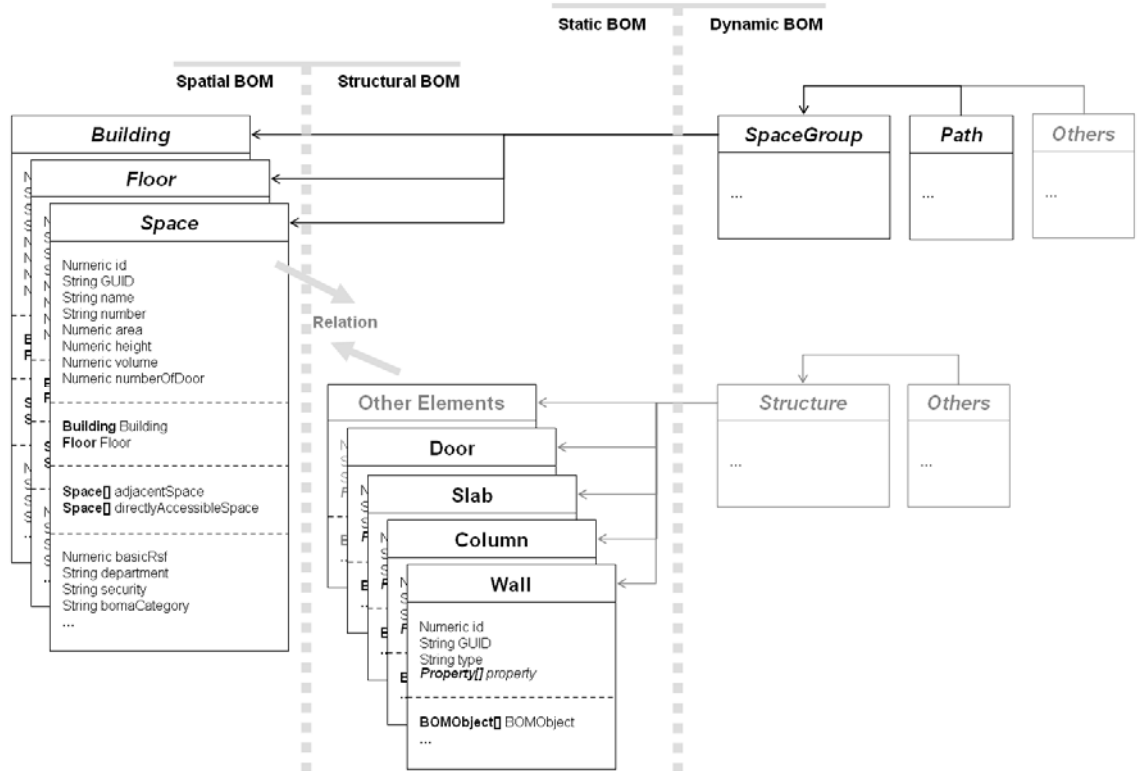


Figure 5.6. Overview of implementation-level BERA Object Model classes.

5.3.1. Static BOM Builder

The BOM handler establishes static BOM data when the model is imported, or before the user input language is parsed or executed. As Figure 3.2 and 4.1 illustrate, BOM is an abstraction of the complex building state focusing on its several “rule and analysis” perspectives. BOM is one of the key concepts to the building environment rule and analysis as the language name literally implies. In the BERA Language Tool implementation, many computed and derived properties have been proposed and implemented for specific purposes, as well as some basic data obtained directly from the given building model. The implementation takes advantages of the input building information, but additional implementation is required to compute some BERA-specific

properties. These are managed by the static BOM builder. The “Building” in Figure 4.2 is the default “container” of any possible instances of BOM, as defined in Figure 5.5: a dot-notation access to the BOM. The static BOM and its properties in Figure 4.2 can be implemented by Java-specific data types such as java double for Numeric, java String for String. The structure can be different from the conceptual structure in Figure 4.2 (front-end), but for efficiency, the implementation described in this dissertation takes the same structure in Figure 4.2. (Also the same relation between Figure 5.7 in the next section and Figure 4.3)

5.3.2. Dynamic BOM Builder

The dynamic BOM builder is in charge of establishing user-defined collection of static BOM. These will be consumed in the design rule & analysis tasks, as is the case with static BOM. For example, in a circulation rule “circulation between A and B should be public”, how can A and B be obtained? General rule-checking software uses space names to acquire them from a given building model. The BERA Language Tool can also support that, but in addition can provide a variety of sophisticated methods using this dynamic BOM definition by users. Not only can their space names be applicable to obtain certain space collections, but also their spatial properties and relations.

This section introduces two major types of dynamic BOM implementation – SpaceGroup and another important sub class of SpaceGroup – Path. They can be originated from the super class ObjectGroup. The main difference between the dynamic BOM and the static BOM is its unlimited instantiability. Users can create any one of the user-defined SpaceGroup or Path instances using pre-determined static BOM in a given building model. Figure 5.7 describes those two classes and their super class –

ObjectGroup. For handling and computing the building circulation-specific properties, there is another static meta-element – Graph. It is used in calculating metric distances and number of turns on the path, and these properties are stored under each Path instance. (Refer to the section 6.3.2 for details on the graph structure and its actual applications in the BERA Language Tool.) SpaceGroup is the super class of Path; therefore Path instances have all properties of SpaceGroup. For example, a Path instance also has properties of SpaceGroup such as ‘numberOfSpace’, ‘height’ and ‘area’.

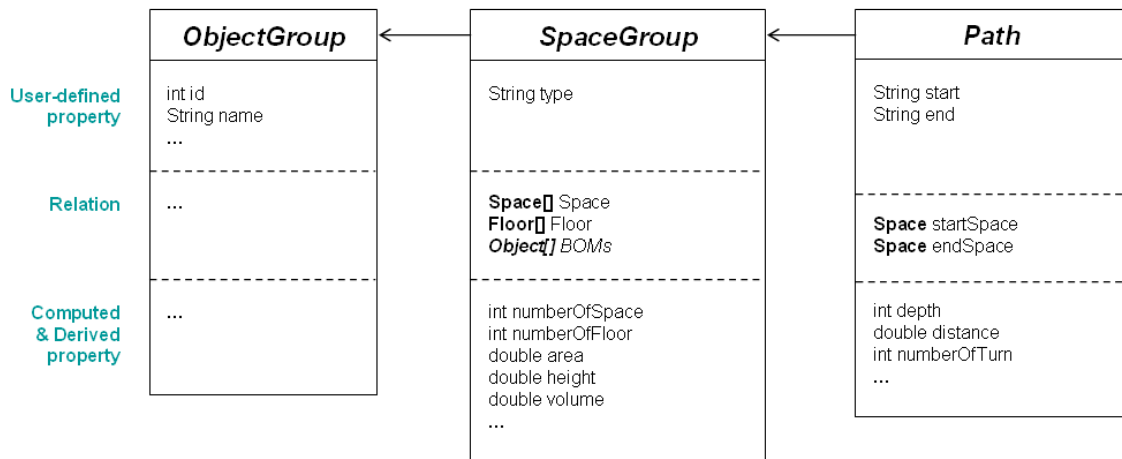


Figure 5.7. Overview of the dynamic BOM and its properties implementation.

The dynamic BOM objects are instantiated by users' BERA Language (The objects named DefBOM in Figure 5.3 and 5.4). The dynamic BOM builder is in charge of instantiating those objects responding to users' ObjectGroup and object definitions. Any dynamic BOM is essentially a derived subset of static BOM. The user's variable name for a SpaceGroup will be a new container for that subset of static BOM. For example, a user-defined ObjectGroup "myOffices" can contain Space objects which are named "office", and this "myOffices" is the container for selected space objects. All the information is loaded in the structure DefBOM – DefCond – DOTExpr structure as shown in Figure 5.4. One of the important features in the dynamic BOM builder is this

kind of object selection algorithm as introduced in Figure 5.8. This returns a series of Boolean results to determine whether the current element *Object[n]* could be selected or not for a given DefBOM. This is also useful to execute the rules. As an example, Figure 5.9 shows this process for the following dynamic BOM definition:

```
Space myOffice {
    Space.name = "office";
    Space.area > 500;
    or Space.Floor.height > 16;
    or Space.department = "office";
}
```

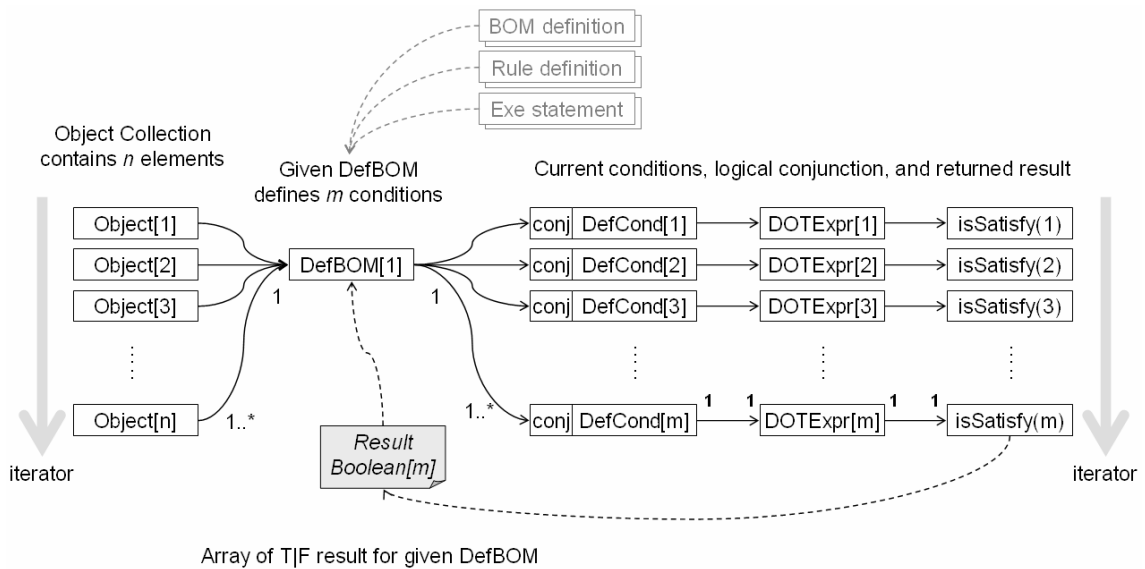


Figure 5.8. Object selection algorithm overview: multiple conditions DefCond in a DefBOM iterate Object Collection and collect its array of Boolean results.

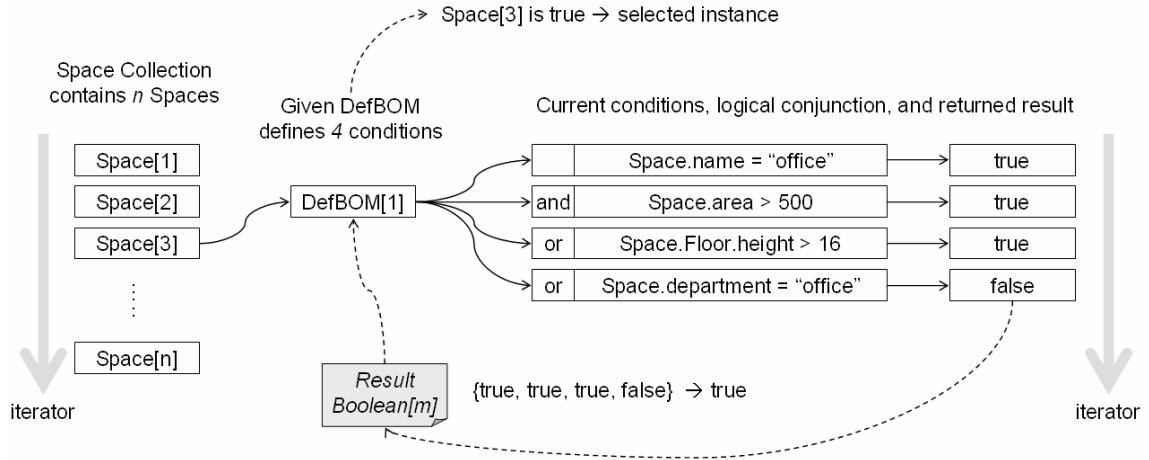


Figure 5.9. Space object selection example for the above program: an instance *Space[3]* is selected because its result is true. A Boolean array {T, T, T, F} returns T because one of the conjunctions is “or” and its value is T¹⁷. (Left to right evaluation)

5.4. BERA Executor

The BERA Language execution statement basically consists of a simple line form – execution commands and their arguments. The fundamental command keyword is ‘get’, as described in the BERA Language design chapter. As it literally means, ‘get’ command retrieves all the BOM and visualizes them based on its arguments. This section focuses on rule checking – in this case, command keywords are user-defined rule names.

User-defined rules basically consist of a variable name, a series of DefCond, and optional nested DefBOM as shown in Figure 5.4. In the implementation level, DefRule and DefBOM have almost the same structure because they are eventually handled by

¹⁷ In this implementation, a nested structure for Boolean array is not yet been developed. Only left-to-right evaluation is allowed in this release, but it will be updated in the next release.

DOTExpr representations and their object selection processes for either defining objects or regulating rule conditions.

The main difference between the definition of BOM and Rule¹⁸ is their container – a dynamic BOM definition basically has default or static BOM, but a rule definition has a dynamic BOM as its container. Similar to the series of examples to describe the object selection algorithm overview, Figure 5.10 shows the object selection process as a rule checking process that is derived from a DefRule. A circulation path collection “p” is the container for this rule definition DefRule, and its DefCond emits the Boolean results through the iteration. This process occurs in the rule execution. A rule execution statement has a certain rule name as a function call, and it delivers user-variable arguments as given object containers. In this example, the container “p” contains *n* number of path instances and this process returns a series of Boolean results to determine whether this instance *Path[n]* is satisfied (selected) by the conditions or not. In the rule checking process, this selected instance means “passed” instance. The BERA executor is in charge of handling the process as well as considering given logical conjunctions on each condition. The simplest result of the rule checking execution is a Boolean – pass or fail; however, the BERA Language Tool provides an entire set of information gathered in this process to users.

¹⁸ Those rule definitions can be stored for reuse in textual format and easily importable/exportable by users through the file I/O and network utilities implemented in the BERA Language Tool.

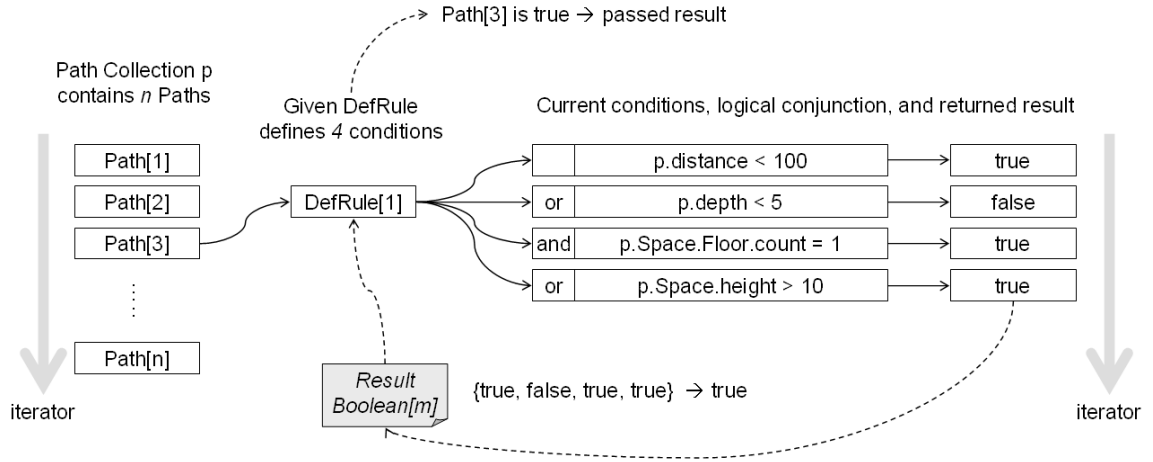


Figure 5.10. Path object selection example: an instance *Path[3]* is passed because its result is true. A Boolean array {T, F, T, T} returns T because one of the conjunctions is “or” and its value is T. (Left to right evaluation)

5.5. BERA Language Tool

The BERA Language Tool is implemented as an integrated development environment of the proposed BERA Language. It is developed as a plug-in software on top of SMC, and runs on the JVM environment. Figure 5.11 is the initial screen of the BERA Language Tool and Figure 5.12 is its editor and console interface. It supports a stand-alone mode and an SMC plug-in mode.

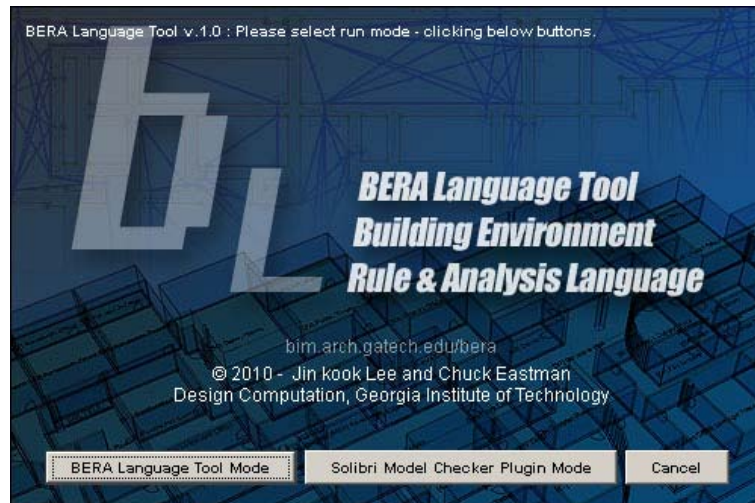


Figure 5.11. BERA Language Tool start-up screen.

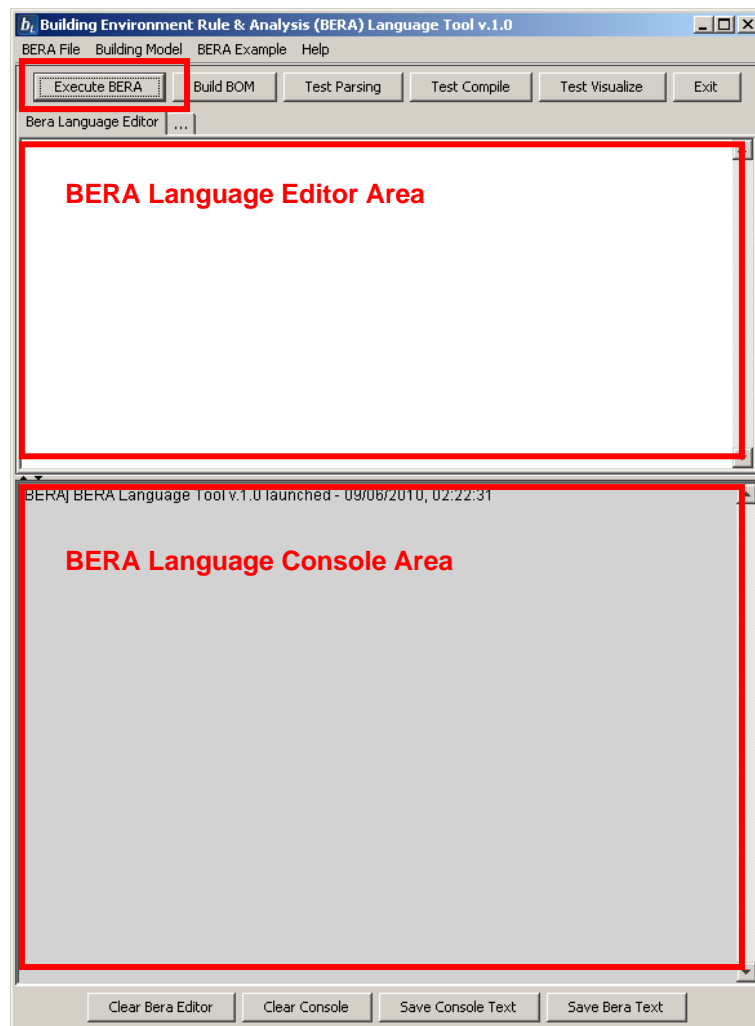


Figure 5.12. BERA Language Tool interface.

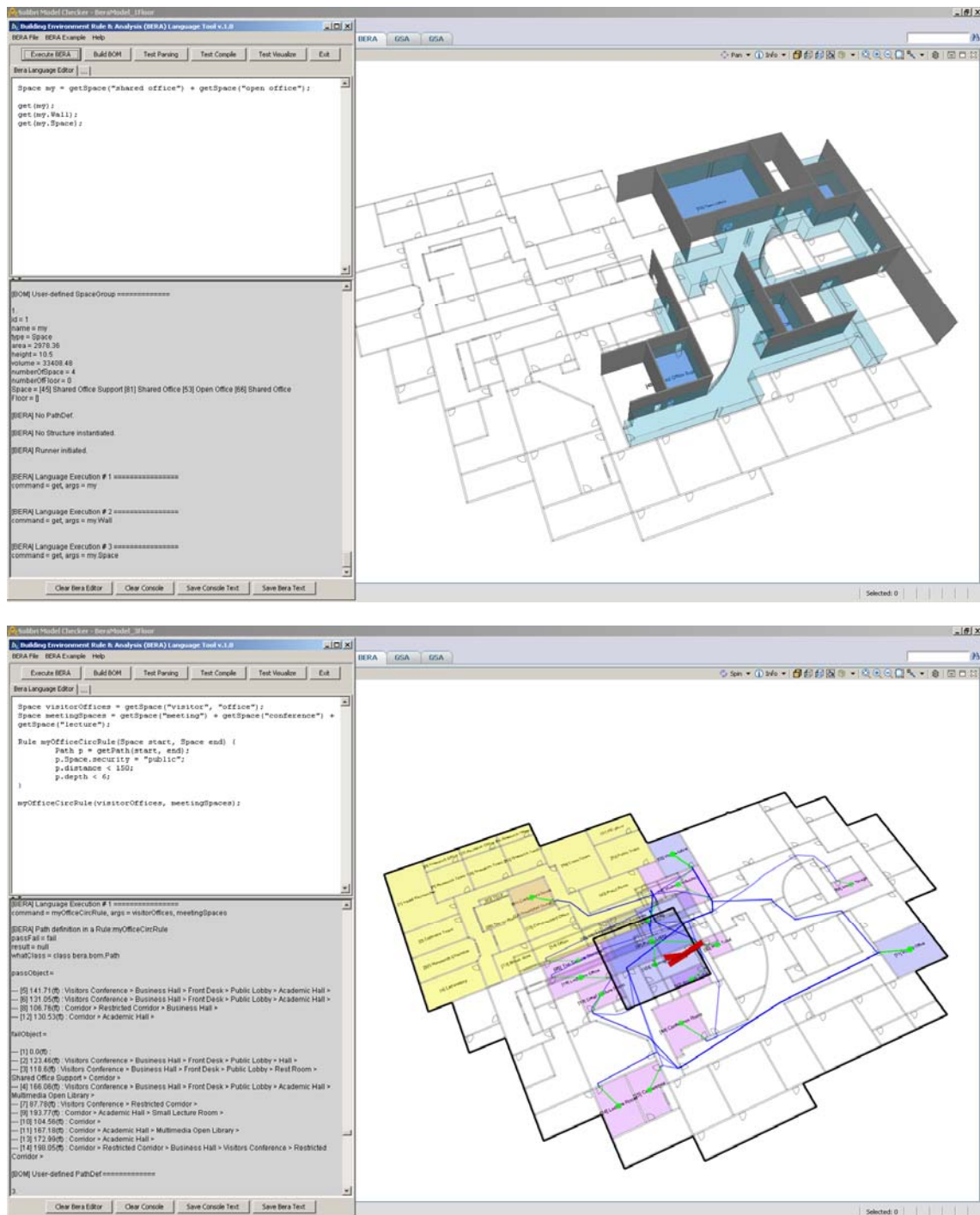


Figure 5.13. BERA Language Tool on top of the BIM platform – SMC.

Figure 5.13 is a screenshot of the plug-in mode of SMC. The window interface on the left is the BERA Language editor and console. Following various iterations of module development, this BERA Language Tool v.1.0 is the initial product-level implementation. The BERA Language-specific features and the BERA Language Tool-related modules are all subject to update in subsequent versions. Appendix D: BERA Language User Manual illustrates detailed features of the tool. The following section deals with extensibility issues.

5.6. BERA Language Extensibility

Similar to the development of other programming languages, the BERA Language development is an open-ended project. Language syntax is technically the main subject of update. This section however emphasizes language extensibility issues focusing on its semantics – front and back-end extensibility. There are two different directions of the BERA Language extensibility:

- 1) Back-end extensibility: Re-targetable BERA Language to support other types of BIM platforms such as BIM authoring tools, model checking tools or simulation tools.
- 2) Front-end extensibility: Extensible BERA Object Model, as well as the issues of BERA Language syntactic/semantic improvement, upgraded BERA Language Tool, etc. BOM extensibility is twofold: lateral extensibility (more building elements responding to the demand of new rules) and vertical extensibility (properties of elements).

5.6.1. Re-targetable BERA Language

Figure 5.14 depicts an overview of data flow in terms of the BERA Language and its target language: Java. As described in this chapter, user BERA Language programs are translated by the reader, and then the interpreter performs a series of internal processes such as lexical, syntactic and semantic analysis, data collection, generation of intermediate representations, and so on. As a result of the translation, a BERA program is re-generated in the target language internally. The series of processes by the BERA listener and BOM handler make the users' input language executable. Issues on the back-end extensibility arise in this phase because they are platform-dependent. How can the BERA Language be transplanted to different platforms? From a software engineering standpoint, the principle of “separation of concerns” [Dijkstra et al, 1982; Parr, 2009] may give clues to the BERA Language back-end extensibility.

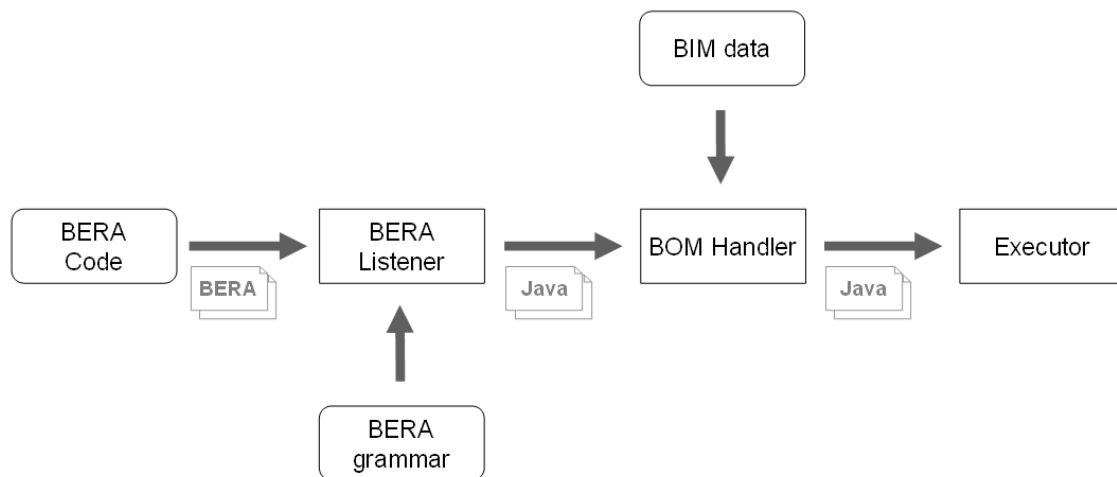


Figure 5.14. A brief data flow diagram to describe the implementation of the BERA Language Tool. A BERA code is translated into a Java code and executed.

As an example translation of BERA code, here is a BERA program to define a dynamic BOM named “bigOffices” which defines two conditions:

```

Space bigOffices {
    Space.name = "office";
    Space.area > 600;
}
get(bigOffices);

```

This is just a 5-line BERA code example, but it should be translated into the target language (in this implementation, it is Java) as follows. This example also demonstrates how BERA Language is effective and easy compared to the Java code below. The example Java code below is an example code in a certain back-end platform.

```

import java.util.ArrayList;
import ...IFCModel;
import ...IFCSpace;
import ...Tools;
// class definition, constructor, method, etc are omitted.

IFCModel model =
(IFCModel)ProductModelHandlingPlugin.getInstance().getCurrentModel();
IFCSpace[] spaces = (IFCSpace[]) model.findAll(IFCSpace.class);
ArrayList<IFCSpace> bigOffices = new ArrayList<IFCSpace>();
for(int i = 0; i < spaces.length; i++) {
    if(Tools.isSameName(spaces[i].name.getStringValue(),
"office") )
        && Tools.sm2sf(spaces[i].area.getDoubleValue()) > 600) {
        bigOffices.add(spaces[i]);
    }
}
if (bigOffices != null) {
    Tools.printInfoToConsole(bigOffices);
    Tools.visualizeOnViewer(bigOffices);
} else {

```

```

Tools.printInfoToConsole("No such objects found.");
}

```

In the actual implementation of the BERA Language Tool, one of the main concerns is that the BERA Language aims to be a re-targetable language considering its extensible capability to other platforms that are developed by different languages and libraries. The target language in this implementation is Java, but other general-purpose languages such as C++ and C# are also available for application using model-driven language translation engines [MDA, 2001; Kent, 2002; Parr, 2009; Parr, 2010b]. This approach makes the BERA Language re-targetable to other platforms¹⁹. For advanced users, the BERA Language Tool also supports its target language directly from the BERA editor, and this enables users to handle a very detailed level of data as well as the API of the target BIM platform. (Refer to the section 6.5.2 as an application example) Figure 5.15 illustrates the overview of re-targetable BERA Listener which translates a same BERA input language to different target languages.

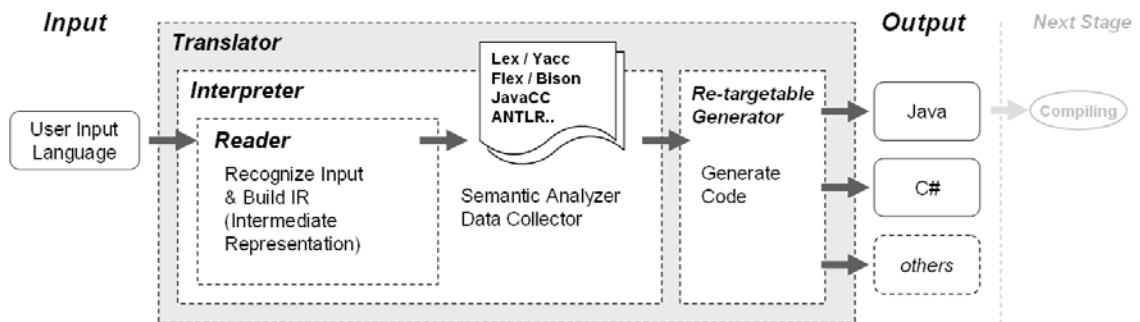


Figure 5.15. Overview of the Re-targetable BERA Listener.

¹⁹ This re-targetable language translation feature is currently dependent upon its parser generator's functionality – ANTLR generator [Parr, 2010a]. It supports a fundamental level of lexer and parser for different target languages. The BERA Language implementation also takes advantages of such utilities thanks to this kind of open tools.

5.6.2. Extensible BERA Object Model

The definition of BOM is open-ended, and the author realized that it is another challenge to define generic and valuable BOM as it grows more detailed. The current BERA Language focuses on the applications of evaluating building circulation and spatial programming with respect to the scope of the research and implementation. In this initial development, spatial BOM is the main point of focus²⁰. Figures 4.2, 4.3 and 5.6 describe object classes that are mainly handled in the current BERA Language and their detailed properties. Additionally, there is a flexible property set named “Property” which allows adding user-defined properties. As reviewed in former sections, the dot-notation access to BOM is intuitively used in both the definition of BOM and rules. There are also several structural building elements available in the current implementation, as shown in Figure 5.16. These have direct relations to the spatial objects such as spaces and floors, and they are instantiatable in the current BOM. For example, Structure is the dynamic BOM similar to SpaceGroup or Path as another sub-type of ObjectGroup. As these are beyond the scope of this dissertation, their properties are not deeply developed yet – some default properties are available directly from the building model. However, they are still building elements that have direct relations to spatial objects. For example, a dot-notation operand `mySpace.Door.width` returns one or many numeric values of the

²⁰ See the section 6.3, 6.4 and 6.5 for demonstrations in these domain applications. The scope of this dissertation is on such domains. However, as an example application of the extensible BOM, the section 6.6 demonstrates a Wall object example. The current version of BERA Language Tool has many features for handling extended BOM as described in Figure 5.16 and still under the development for further applications.

width of the doors in a group of spaces named `mySpace`, as it implies. (Refer to Figure 5.5 and Appendix D)

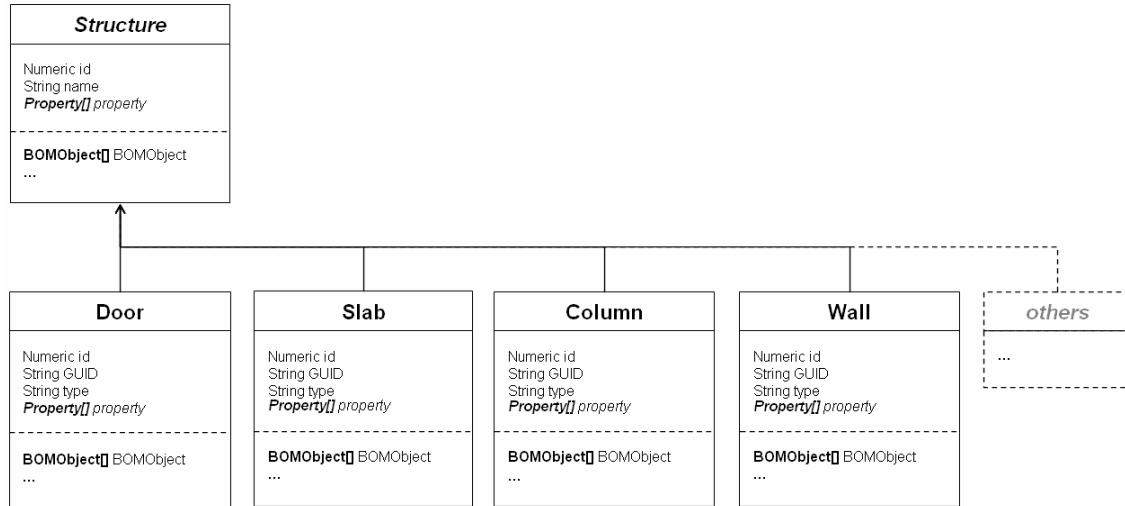


Figure 5.16. An example of extended building objects: *Structure* is an example class of the dynamic BOM of these structural building objects. For example, the object group “all walls and slabs of the basement floor” or “all exterior walls” can be dynamically instantiated as an instance of *Structure*, by the user. It will be used for the rule checking, analysis, or just for various visualizations.

Lateral extensions such as structural building elements (as shown in the examples in Figure 5.16) and vertical extensions (such as additional properties for existing BOM objects) are good examples of the open-endedness of the BERA Language. Figure 5.17 illustrates this two-way extensibility of the BOM development. In the static and dynamic BOM described in figures 4.2 and 4.3, many computed and derived properties have been proposed and implemented for the following purposes: evaluating building circulation and spatial programming. There are many challenging issues in both lateral and vertical extensibility according to the domain and scope of the ‘building environment rule and analysis’. Therefore, BOM could have multiple model views with different conventions.

In other words, the current implementation and applications described in the following chapter is one of the model views of BOM for evaluating building circulation and spatial program. The goal is to provide easy access to the concepts of a domain, for rule checking.

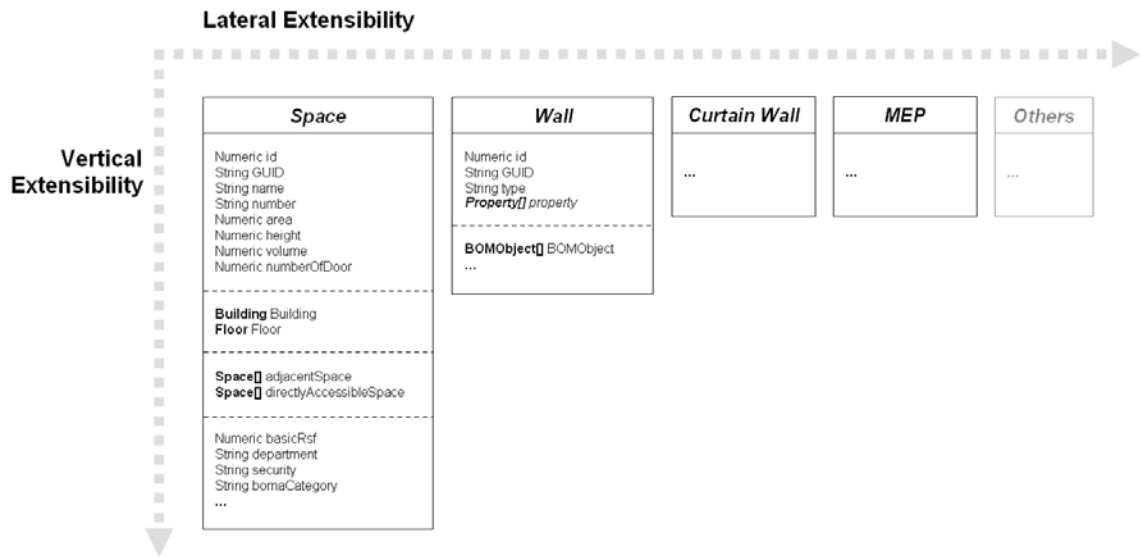


Figure 5.17. Two-way extensibility of the BERA Object Model.

CHAPTER 6

APPLICATION AND EVALUATION

6.1. Overview

This chapter reports on the results of the application and evaluation of the BERA Language. The BERA Language has been designed and defined for building environment rule and analysis as BERA literally means. However, it was implemented with the initial focus of application for evaluating building circulation and spatial programming. The BERA Language Tool version 1.0 has been implemented, and it will be described in detail using various examples (also refer to Appendix D: BERA Language User Manual). By using selected rules and analysis examples, the BERA Language will be evaluated with respect to its current purpose and scope. Also this chapter will attempt to show how the BERA Language is easy, effective on actual building design reviews, based on pragmatic example programs regarding real-world rules. Compared to the pre-determined software-driven methods, the proposed BERA Language Tool as a user-driven method shows higher flexibility and fidelity to the domain-specific problems. The term ‘problems’ here include not only general building design problems but also domain-specific problems such as the rule checking issues in building circulation and spatial program, as the section 1.1.3 described the scope and domain of this dissertation. In other words, ‘fidelity’ means that the BERA Language design should be capable of handling the issues on the general problems of building environment rule and analysis (BERA), and the focus of ‘fidelity’ in this initial design and implementation is on the problems of building circulation and spatial program. In terms of its extensibility, the section 5.6.2

demonstrated how BERA Language is capable of handling different type of problem domains.

The BERA Language Tool has been carefully implemented and tested. The ultimate testing will come from language users, and the open-ended testing and support arrangement is planned. This chapter attempts to demonstrate its capability for handling domain-specific problems regarding the applications of spatial programming and building circulation. Therefore, this chapter places emphasis on describing the key aspects of the BERA Language for the application and evaluation of the rules associated with such issues. As described in the language design chapter, this research does not attempt to analyze or deal with the entire set of implicit rules, design guides or codes worldwide. However, they are still important references because the BERA Language should be expressive enough to reflect them in building environment rule and analysis tasks. The author and his team have been involved in several design rule-checking software development projects [Eastman et al, 2009c; Lee, J-K, 2010b; Lee, J, 2010; Sanguinetti et al, 2010]. Reflecting on empirical knowledge from actual projects, this chapter is organized as follows to effectively describe the BERA Language application and evaluation:

- 1) To review the issues from real-world rules and existing software-driven methods regarding the building circulation and spatial programming,
- 2) To describe important capabilities of the BERA Language to handle rule conditions: how the BERA Language and BOM handle building objects, their properties, operations, and values, and
- 3) To describe the capability of the BERA Language in terms of its expressiveness for addressing the complex state of a building and rules as the collection of rule conditions.

This chapter describes how the BERA Language is capable of analyzing building models in terms of its functionality to handle building elements and their properties. The following sections (6.3 through 6.5) describe the actual applications for evaluating building circulation and spatial programming using several example BERA programs and their results.

6.2. Real-world Rules and Analysis

6.2.1. Software-driven Methods for Handling Real-world Rules

Start	Required	Destination	Transition Conditions	
Name: Press / Media Room		Name: Bankruptcy courtroom,Us...	Security Level: public,Usage: circulation,Route Length: 100.00 m,Verti...	...
Name: Press / Media Room		Name: Magistrate courtroom,Us...	Security Level: public,Usage: circulation,Route Length: 100.00 m,Verti...	...
Name: Trial Jury Room	ALL Name: Soundlock	Name: -	Security Level: public,Usage: circulation,Vertical Access: allowed	...
Name: Grand Jury Hearing Room	ALL Name: Entry Security Station	Name: -,Usage: circulation	Security Level: restricted,Usage: circulation,Vertical Access: allowed	...
Name: Circuit Librarian		Name: Library Entry/Lobby	Security Level: restricted,Usage: circulation,Route Length: 100.00 m,...	...
Name: Circuit Librarian		Name: Library Circulation Area	Security Level: restricted,Usage: circulation,Route Length: 100.00 m,...	...
Name: Circuit Librarian		Name: Reference/Card Catalog ...	Security Level: restricted,Usage: circulation,Route Length: 100.00 m,...	...
Name: Circuit Executive's Office		Name: -,Usage: circulation	Security Level: restricted,Usage: circulation,Vertical Access: allowed	...
Name: Senior Staff Attorney		Name: -,Usage: circulation	Security Level: restricted,Usage: circulation,Vertical Access: allowed	...
Name: Senior Staff Attorney		Name: Library Entry/Lobby	Security Level: restricted,Usage: circulation,Route Length: 100.00 m,...	...
Name: Senior Staff Attorney		Name: Court of Appeals Clerk o...	Security Level: restricted,Usage: circulation,Route Length: 100.00 m,...	...

Figure 6.1. Table-based parameters example for the US Courthouse circulation rules.

In commercial rule checking software, rule statements are translated and handled in a certain type of data structure: mostly in a tabular structure of parameters [Eastman et al, 2009; Lee, J et al, 2010]. Figure 6.1 shows an interface to handle such table-based parameters. This example represents the US Courthouse circulation rules [USCDG, 1997; 2007] in SMC [Solibri, 2010]. Figure 6.2 shows one of its parameter editing interfaces. A row contains a single rule, and each column contains its conditional parameters. For example, the first row represents a circulation rule: “A circulation from press/media room to the courtrooms should be public”. The column named ‘Start’ contains a space name

‘press/media room’, the column ‘Destination’ contains multiple courtrooms named ‘bankruptcy courtroom’, ‘district courtroom’, etc, and the column ‘Transition Conditions’ contains multiple circulation conditions such as ‘Security Level: public’. Using this structure, over hundreds of ‘translated and parameterized’ rules can be evaluated by the software for a given building model.

Although these table-based parameters handle the rules in a certain situation fairly robustly, the rules represented in this format have limitations. For instance, transitional conditions for circulation paths are pre-defined by a set of properties: security level, usage, distance, direct access, and vertical access. Figure 6.2 shows the interface to control these properties. Another important role of the property is that it can be used in collecting objects, as shown in the example in Figure 6.3. It is systematically equivalent to the example in Figure 6.2 in terms of their Object – Property – Operator – Value relations. The biggest limitation in the table-based parameters method is on its domain-specific and user interface-dependent features, as same as other applications. It is the limitation of this kind of software-driven method, in terms of its extensibility; e.g. there is a very limited way to add additional properties or operations within this table-based parameters method. However, the language-driven method can overcome this limitation because it is based on a generalized model – in this case, it is the BERA Object Model derived and computed from a given building model. As shown in Figure 4.2 and 4.3, fairly many domain-independent object properties (including some domain-specific properties) are available to users for the same task. The rules are maximally extensible with ease when the BOM is extended laterally or vertically. (Refer to the section 5.6) Moreover, the BOM structure in this initial implementation has some user-assignable properties. (Refer to the section 5.3) More user-manageable properties are desirable for the further developments of BERA Language Tool in terms of its BOM extensibility.

On/Off	Condition	Value
<input checked="" type="checkbox"/>	Security Level	restricted
<input checked="" type="checkbox"/>	Usage	circulation
<input checked="" type="checkbox"/>	Route Length	100.00 m
<input type="checkbox"/>	Direct Access	
<input type="checkbox"/>	Vertical Access	

Figure 6.2. Table-based parameters example for one of the conditions shown in Figure 6.1.



Enabled	Component	Property	Operator	Value
<input checked="" type="checkbox"/>	 Space	Name	=	Break Room
<input checked="" type="checkbox"/>	 Space	Area	>	500.00 sq ft

Figure 6.3. Table-based parameters example for collecting space object components using parameters.

6.2.2. Handling Rules by the BERA Language

As the introduction and language design chapter described, the development of the BERA Language basically aims to overcome limitations from software-driven methods. By using the BERA Language, many more properties are available and dynamically applicable as many as BOM has: for example, spatial depth, number of turns, area, volume, height, and even other related types of building elements such as number of windows or area of windows on the walls along the path, etc are available to users, but the parameter-driven method shown in Figure 6.1 cannot support them so far²¹. These are

²¹ In the table in Figure 6.1, total 183 rows are defined for the circulation rules. In the BERA program for the same rule checking, there are only 24 different rule definitions for handling those rules because the space names can be defined in execution statements as given user arguments rather than in the rule definitions. Execution statements can directly contain space names for start, required and target spaces, but BOM definition statements handle them more precisely and flexibly for different building models.

derived and computed from a given building model using the BOM Handler (refer to the section 5.3). Moreover, these properties are subject to extension, as introduced in the former section: vertical extensibility of the BERA Object Model. The more the properties are developed, the more sophisticated rules that can be evaluated without additional time-consuming implementation. The author believes that this is one of the idealistic ways to make the BERA Language with high fidelity to the problems of real-world rules, rather than analyzing and customizing all the rules in the world.

One of the fundamental problems is that all real-world codes, rules or design guides are written in natural language for humans, not in the BERA Language or any other type of explicit forms for computers. There will be some expressions that are nuanced and subject to interpretation. Moreover, design guides, rules, or codes are managed by different organizations and stakeholders in very different ways. Publicly available design guides, rules, and code examples that have been reviewed by the author and his team are: [AIA, 1997; US Access Board, 2002; City of New York, 2004; NFPA, 2006; USCDG, 1997; 2007; ICC, 2010; WBDG, 2010]. Among these, the US Courts Design Guides [USCDG, 1997; 2007] have been carefully reviewed and translated into the table-based parameters as shown in the previous section. The issues including rule translation and formalization are beyond the scope of this research. The detailed issues on analyzing circulation rules and generating table-based parameters are described in [Lee, J, 2010]. Instead, reflecting the author and his team's efforts on the research and development project [Eastman et al, 2009], this chapter focus on describing how the proposed BERA Language has the needed semantic capability to express various kinds of rules and conditions.

Real-world rules are complicated and subject to discussion considering the multiple involved perspectives that are mostly based on qualitative issues. One of the proposed solutions to this problem is in the newly proposed abstraction of the state of a building – the BERA Object Model (BOM). The BERA Language basically deals with

building objects, their properties, operators, and values for the building environment rules and analysis purposes. By using BERA Language, an operand which consists of a dot-notation access to the BOM, operator, and a value is technically in charge of representing a rule or one of its conditions. In other words, this enables the rule statements to be broken down to the structured and operable expressions for both humans and computers.

In the following sections, example programs will attempt to demonstrate the questions in the BERA Language Design chapter. They can be summarized as follows:

- 1) The BERA Language supports various definition methods of dynamic BOM. BERA Language supports dot-notation based conditional definitions for user-defined BOM, while the name string-based matching is the only way in current software-driven method.
- 2) The dot-notation access to BOM can be used in both BOM definitions and rule definitions. In each definition, multiple conditions can be described with expressive statements such as logical conjunctions: ‘and’ and ‘or’.
- 3) The BERA Language supports various operational statements for handling both the complexity of design rules, and the complex relations of space objects and properties. They are logic operations, logic values, recursion, negation, and so on. Also the BERA Language Tool proposed and implemented for this dissertation also supports the target programming language (in this implementation, Java) for advanced users to access lower level data as well as direct access to the BIM platform’s API.

The BERA Language is by no means complete. However, the BERA Language Tool has demonstrated that it enables users to analyze building models in much easier way than general purpose programming languages, and more powerful than pre-defined application interfaces such as table-based parameters that were reviewed in the previous section. As the BERA Language is designed to be an easy and powerful domain-specific

language, the BERA Language program examples in the following sections will demonstrate its fidelity to the actual issues.

6.3. Application for Handling Building Objects

For the application and evaluation, a variety of building models have been tested using the BERA Language Tool v.1.0, including some building models that are generated by professional architects for actual building projects. However, this chapter describes the test results using a single model due to some building information non-disclosure issues and consistency of tests. A test building model is illustrated in Figure 6.4. It is a test-purposed concept design model which has 104 spaces, 3 floors, 115 doors, 17 columns, 185 walls, and the gross area is 45,336 SF. The model was simply yet carefully modeled with associated object-properties, and there are no differences from the models commonly used in actual projects. This chapter describes the examples based on this model. Each program example is described in the table which has three rows: first shows the BERA program example, second shows its visualization result captured from the BIM platform (SMC), and the last row displays the textual result returned from the BERA console (some are omitted where it is too long). In the former chapter, Figure 5.14 and 5.14 shows the interface of the tool: BERA Language editor, console, and model view interface in SMC.

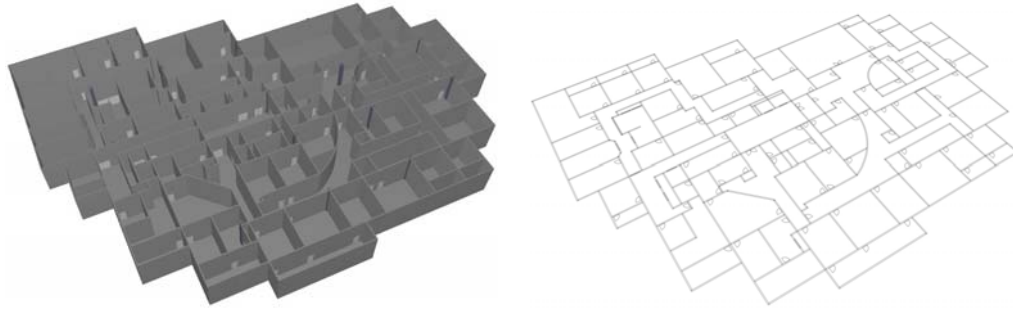


Figure 6.4. A test model for running BERA Language programs in this chapter²².

6.3.1. Static Building Elements

Before dealing with complex BERA program examples, here are some basics on static building objects such as spaces, floors, and their container - the building. Because the BERA Language Tool does not attempt to change the building model itself, their associated properties can be statically established when a given building model is loaded into the system, until it is closed by the user. This section describes some BERA program examples to handle and display such static building objects, as same as the static BOM definition in Figure 4.2.

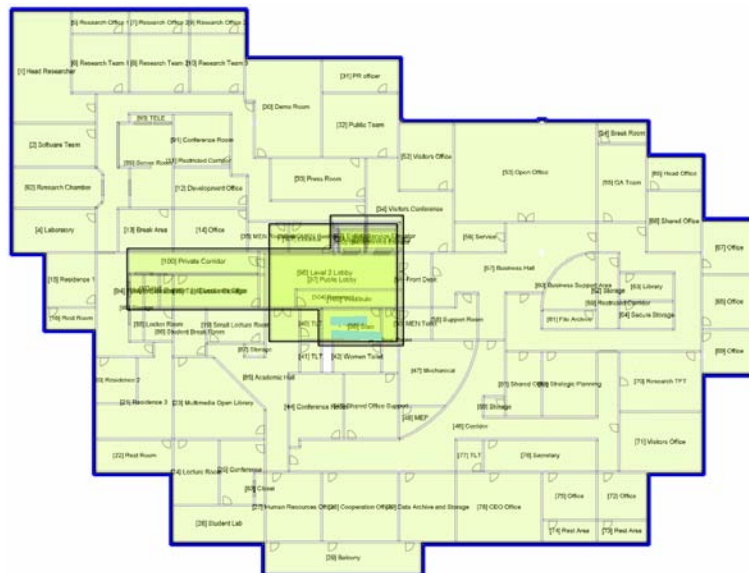
Table 6.1 shows how to acquire building, floor, and space objects using a simple ‘get’ command. For `get(Building)`, a textual information set defined in BOM is displayed in the BERA console. The execution statement `get(Floor)` generates textual information and graphical visualization of floor’s boundary polygons (the first floor is

²² The test building models’ screenshot images in this chapter were captured from the BERA Language Tool v.1.0, based on the BIM platform Solibri Model Checker® v6. The model was generated by Autodesk Revit® 2010, and exported into IFC 2X3. For better visibility, two floors were intentionally modeled in small size, and they were dropped where the vertical circulation and its visualization was not important.

colored in blue). The statement `get (Space)` displays entire spaces' properties in textual format and visualizes their spatial boundary polygons filled with a color. The BERA console prints out entire set of information, but the tables in this chapter only shows subset of them because sometimes they are too long ('...' means the omission of some repeated data from the tables). Also the result displayed in these examples is offered to review the detailed information of the test model used in this chapter.

Table 6.1. To retrieve static building objects using: `get (args)`.

```
get (Building);
get (Floor);
get (Space);
```



[BERA] Parsing bExeStat =====

```
1.
exeCommand = get
args = Building
argsType1 = Building
argsType2 =
2.
```

```
exeCommand = get
args = Floor
argsType1 = Floor
argsType2 =
3.
exeCommand = get
args = Space
argsType1 = Space
argsType2 =
```

```
[BERA] Language Execution # 1 =====
command = get, args = Building
```

```
[BOM] Building =====
```

```
1.
id = 1
fileName = BeraModel_3Floor
numberOfFloor = 3
numberOfSpace = 104
numberOfDoor = 115
numberOfSlab = 5
numberOfColumn = 17
numberOfWall = 185
area = 45336.4
totalNetArea = 41664.88
height = 31.0
elevationHeight = 47.0
volume = 875479.32
firstFloor = [2] Level 1
designPhase = Late Concept Design
bimDesignTool = Autodesk Revit Architecture 2011
ifcVersion = IFC2X3
buildingGrossArea = 45336.4
structuredParkingArea = 0.0
mepArea = 852.66
skinArea = 42985.36
externalWallArea = 2331.48
biggestFloorArea = 40653.88
...
```

```
[BERA] Language Execution # 2 =====
command = get, args = Floor
```

```
[BOM] Floor =====
```

```
1.
id = 1
GUID = 3_K4qH1dT7kQaYapBFHa8k
name = Level 1
area = 40653.88
totalNetArea = 37492.56
height = 20.0
elevationHeight = 0.0
volume = 813077.6
```

number = 1
Space = [bera.bom.Space@d86395, ... bera.bom.Space@96a71f]
Slab = [bera.bom.Slab@1481cb6]
...

[BERA] Language Execution # 3 =====
command = get, args = Space

[BOM] Space =====

1.
id = 1
GUID = 3glzoCOwD6aRpAaskFPtU6
name = Storage
number = 90
area = 111.1
height = 10.0
volume = 1111.0
Building = bera.bom.Building@1547ec9
Floor = bera.bom.Floor@b8b6e9
basicRsf = 130.53
department = Residence
security = Secure
bomaCategory = Office

2.
id = 2
GUID = 2\$MVzGbqDDDx2L1crlzBrm
name = Closet
number = 83
area = 36.33
height = 10.0
volume = 363.3
Building = bera.bom.Building@1547ec9
Floor = bera.bom.Floor@b8b6e9
basicRsf = 46.94
department = Education Center
security = Public
bomaCategory = Office

...

92.
id = 92
GUID = 3jj0lDqC1F68GyiajiLG2A
name = Locker Room
number = 88
area = 117.15
height = 10.0
volume = 1171.5
Building = bera.bom.Building@1547ec9
Floor = bera.bom.Floor@b8b6e9
basicRsf = 132.08
department = Education Center
security = Secure

6.3.2. Static Meta-Element Example: Graph

In current BERA Language Tool implementation, there is another important static meta-element – graph. This section describes this meta-element that has an important role in domain-specific issues: building circulation representation and computing traversal conditions such as distance, number of turns, spatial topology, and so on. (This has been developed by Georgia Tech team, including the author.) In terms of the definition of BOM, currently it is an external object, however in the future development of the BERA Language and its Tool, this kind of element has a potential to be a part of the BOM structure.

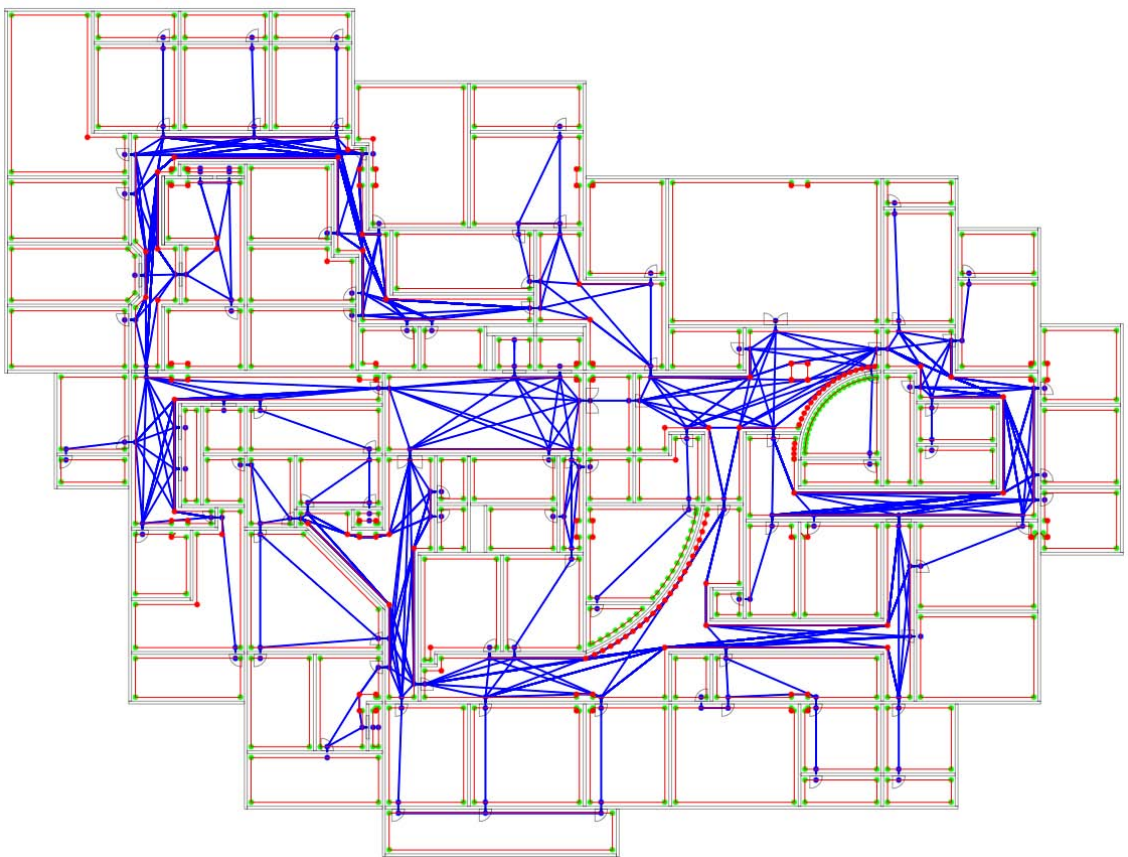
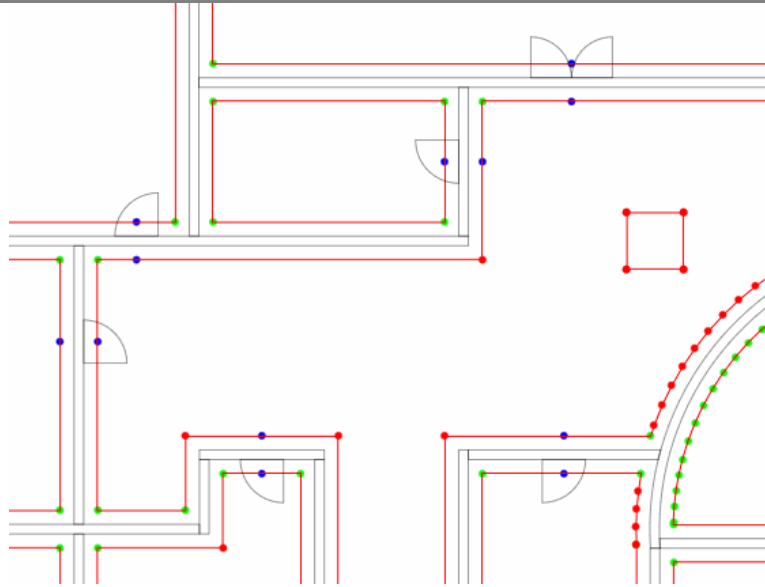
For the circulation rules and analysis, graph components should be prepared for dynamically computing and visualizing the circulation path instances. The class ‘Path’ is a dynamic BOM as defined in Figure 4.3, but the graph components for representing paths can be statically generated by the given building model because they can be pre-defined by static building elements such as spaces, space boundaries, doors, stairs, walls, columns, etc, as well as spatial topologies. Briefly, the graph structure implemented in the BERA Language Tool is a computational method for measuring walking distances and visualizing paths within buildings based on a length-weighted graph structure for a given building model. It is mainly determined by a given building’s spatial topology and geometry, and it returns consistent and accurate scalar quantities. It takes into consideration people movement patterns, reflecting that people tend to walk along the shortest, easiest, and most visible paths. It has been implemented by the author and the team at Georgia Tech based on open libraries, SMC platform, and theoretical literatures

[Peponis et al, 1998; Gross, 1998; Dym, 1998; Werner et al, 2001; Duckham et al, 2003; Zhi et al, 2003; Kannala, 2005; Goodrich 2005; Dijkstra, 2010; Solibri, 2010]. For more detail of the graph components and their computing algorithms, refer to [Lee, J-K et al, 2010a; Lee J, 2010]. The graph used in the BERA Language Tool is a building circulation-specific feature as one of the applications of BERA Language, within the scope of this dissertation. It facilitates this kind of spatial circulation-specific purposed tasks, and will be available in different types of graph structures for supporting other domain-specific problems as an abstraction of the complicated building elements.

Table 6.2 shows two graph component related commands, and they only visualize graphs without textual print-outs in BERA console because they are a kind of meta-data in this stage. It is meaningful when actual path instances are populated, as shown in later part of this chapter for circulation paths. In that case, BERA console prints out all circulation-associated information.

Table 6.2. To retrieve static graph elements using: `get(args)`.

```
get(Graph);
get(GraphComponent);
```



6.3.3. Dynamic Object Group Definition: SpaceGroup

The space object instances under SpaceGroup can be instantiated dynamically and unlimitedly, as described in the section 4.3 and Figure 4.1. This is why SpaceGroup is a dynamic BOM (see Figure 5.9) to represent any spatial objects that users are interested in. A dynamic BOM object can be instantiated by users in several ways. A formal long-form style for generating an instance of SpaceGroup is described in Table 6.3, but the short forms are also available as shown in the BERA Language definition chapter using ‘getSpace’ keyword with its arguments and its combinatorial forms. For defining an instance of SpaceGroup, user should specify its BOM type, assign a user-variable name, and describe its conditions. The type of BOM is semantically always SpaceGroup, but simply Space also leads to same result because they are all still space objects. Other types are Floor and Path in the scope of this implementation.

The BERA Language Tool supports multiple other ways for defining dynamic BOM as described in the Appendix D: BERA Language User Manual. For example, in the name-based mapping, BERA Language Tool supports an internal method called object name-based mapping method for establishing building object semantics. In this implementation, its focus is on establishing space object semantics by given space names. That is, even if user type simply “office”, BERA Language tool retrieve not only “office” but also other office spaces such as “office 10”, “head office”, “open office”, “shared offices”, etc that are semantically mean “office”. This method is based on a series of useful open methods such as sub-string algorithm, string matching algorithm by levenshtein distance [Levenshtein, 2010], etc. This is a BERA Language-specific feature for retrieving maximum object semantics from object names. Based on several actual projects done by the author and his team, it has been demonstrated that it is very pragmatic in actual projects. The author and his team have researched such spatial object

semantics issues in actual projects. Refer to [Lee, J-K, 2010b] for details on the space object semantics issues.

Table 6.3. To instantiate a certain group of space objects named ‘midOffice’ using a typical dynamic BOM definition method. It can be simply retrieved by ‘get’ command with ‘midOffice’ as its argument.

```
Space midOffice {  
    Space.area > 600;  
    Space.area < 900;  
    Space.height > 9;  
    Space.name = "office";  
    Space.name != "shared";  
}  
  
get(midOffice);
```



[BERA] Parsing bBOMDef =====
1.

```

name = midOffice
BOMType = Space
condition = Space.area > 400;
    Space.area < 900;
    Space.height > 9;
    Space.name = "office";
    Space.name != "shared";
Definition BOM Prop:
lop =
operand_left = Space.area
operator = >
operand_right = 400
- BOM1 = Space
- property = area
lop =
operand_left = Space.area
operator = <
operand_right = 900
- BOM1 = Space
- property = area
lop =
operand_left = Space.height
operator = >
operand_right = 9
- BOM1 = Space
- property = height
lop =
operand_left = Space.name
operator = =
operand_right = "office"
- BOM1 = Space
- property = name
lop =
operand_left = Space.name
operator = !=
operand_right = "shared"
- BOM1 = Space
- property = name

[BERA] Parsing bExeStat =====
1.
exeCommand = get
args = midOffice
argsType1 = BID
argsType2 =

[BOM] User-defined SpaceGroup =====
1.
id = 1
name = midOffice
type = Space
area = 3416.0
height = 10.33
volume = 35080.66
numberOfSpace = 6
numberOfFloor = 0

```

```
Space = [14] Office [27] Human Resources Office [78] CEO Office [28]
Cooperation Office [52] Visitors Office [71] Visitors Office

[BERA] Language Execution # 1 =====
command = get, args = midOffice
```

6.3.4. Floor Definition

Floor is another important spatial object in static BOM definition (See Figure 4.2). Similar to the definitions of spaces, given building floors can be defined by users as an example described in Table 6.4. The BERA Language Tool only visualizes their floor boundary polygons when they are called by users, but the BERA Console Area displays all associated information for the floors: area, height, volume and related objects.

Table 6.4. To instantiate two user-defined floors named ‘bigFloor’ and ‘upperFloor’ and display them using get command. They are visualized in the system using their boundary polygon, and displayed in the BERA console with associated information in current BERA Language Tool.

```
Floor bigFloor = getFloor(Floor.area > 3000);

Floor upperFloor {
    Floor.number > 1;
    Floor.area > 1000;
    Floor.name != "base";
}

get(bigFloor);
get(upperFloor);
```



```
[BERA] Parsing bBOMDef =====
1.
name = bigFloor
BOMType = Floor
condition = getFloor(Floor.area > 3000)
Definition BOM Prop:
lop =
operand_left = Floor.area
operator = >
operand_right = 3000
- BOM1 = Floor
- property = area
2.
name = upperFloor
BOMType = Floor
condition = Floor.number > 1;
          Floor.area > 1000;
          Floor.name != "base";
Definition BOM Prop:
lop =
operand_left = Floor.number
operator = >
operand_right = 1
- BOM1 = Floor
- property = number
lop =
operand_left = Floor.area
operator = >
operand_right = 1000
- BOM1 = Floor
- property = area
lop =
operand_left = Floor.name
```

```

operator = !=
operand_right = "base"
- BOM1 = Floor
- property = name

[BERA] Parsing bExeStat =====
1.
exeCommand = get
args = bigFloor
argsType1 = BID
2.
exeCommand = get
args = upperFloor
argsType1 = BID

[BOM] User-defined SpaceGroup =====

1.
id = 1
name = bigFloor
type = Floor
area = 40653.88
height = 20.0
volume = 813077.6

2.
id = 2
name = upperFloor
type = Floor
area = 2503.72
height = 11.0
volume = 27540.92

[BERA] Language Execution # 1 =====
command = get, args = bigFloor

[BERA] Language Execution # 2 =====
command = get, args = upperFloor

```

6.3.5. Path Definition

The BERA Language Tool defines Path instances using SpaceGroup and its associated graph structure derived from the spatial topology and geometry, as defined in the section 4.3 and Figure 5.9. In terms of its visualization, a circulation path can be

represented in several ways: a series of spaces, topological graph structure, metric graph structure, etc. The BERA Language Tool uses a metric graph structure for calculating distances and representing building circulation. It was developed by the author and his team as a plug-in of SMC [Lee, J-K et al, 2010a]. Table 6.5 shows an example of instantiating Path. Similar to ‘getSpace’ or ‘getFloor’, a method named ‘getPath’ generates path instances between two dynamically generated space groups, named ‘start’ and ‘end’. Also ‘get’ command displays the instantiated path objects as shown in Table 6.5. This is an example of the instantiation of Path – the section 6.5 will describe how the definition of Path is applicable for evaluating building circulation rules.

Table 6.5. To instantiate two different SpaceGroup ‘start’ and ‘end’ first, and put them into another dynamic BOM (Path) definition named ‘myPath’. The BERA Language Tool computes circulation paths between two space groups. As shown in this Table, ‘start’ has 3 spaces and ‘end’ contains 4 space instances, therefore total 12 circulation path instances are populated by this definition. The system visualizes all of their paths with graph structures and highlighted start and end spaces.

```
Space start = getSpace("laboratory") + getSpace("lobby");
Space end {
    Space.area > 600;
    Space.Floor.number > 0;
    Space.Floor.height > 10;
    Space.name = "office";
}

Path myPath = getPath(start, end);
get(myPath);
```



```
[BERA] Parsing bBOMDef =====
1.
name = start
BOMType = Space
condition = getSpace("laboratory") + getSpace("lobby")
Definition BOM Prop:
lop =
operand_left = Space.name
operator = =
operand_right = "laboratory"
- BOM1 = Space
- property = name
lop =
operand_left = Space.name
operator = =
operand_right = "lobby"
- BOM1 = Space
- property = name
2.
name = myPath
BOMType = Path
condition = getPath(start, end)
Definition BOM Prop:
lop =
operand_left = start
operator = Path
operand_right = end
3.
name = end
BOMType = Space
condition = Space.area > 600;
```

```

        Space.Floor.number > 0;
        Space.Floor.height > 10;
        Space.name = "office";
Definition BOM Prop:
lop =
operand_left = Space.area
operator = >
operand_right = 600
- BOM1 = Space
- property = area
lop =
operand_left = Space.Floor.number
operator = >
operand_right = 0
- BOM1 = Space
- BOM2 = Floor
- property = number
lop =
operand_left = Space.Floor.height
operator = >
operand_right = 10
- BOM1 = Space
- BOM2 = Floor
- property = height
lop =
operand_left = Space.name
operator = =
operand_right = "office"
- BOM1 = Space
- property = name

[BERA] Parsing bExeStat =====
1.
exeCommand = get
args = myPath
argsType1 = BID
argsType2 =

[BOM] User-defined SpaceGroup =====

1.
id = 1
name = start
type = Space
area = 2069.12
height = 10.67
volume = 22545.8
numberOfSpace = 3
Space = [37] Public Lobby [4] Laboratory [96] Level 2 Lobby

3.
id = 3
name = end
type = Space
area = 3871.37
height = 10.5
volume = 42338.58

```

```

numberOfSpace = 4
Space = [78] CEO Office [28] Cooperation Office [53] Open Office [71]
Visitors Office

[BOM] User-defined PathDef =====

2.
id = 2
name = myPath
start = start
end = end
numberOfStartSpace = 3
numberOfEndSpace = 4
numberOfPaths = 12
startSpace = [37] Public Lobby [4] Laboratory [96] Level 2 Lobby
endSpace = [78] CEO Office [28] Cooperation Office [53] Open Office
[71] Visitors Office
Path = [1] 100.48(ft) [2] 66.39(ft) [3] 54.56(ft) [4] 119.68(ft) [5]
222.17(ft) [6] 165.61(ft) [7] 176.25(ft) [8] 241.37(ft) [9] 159.89(ft)
[10] 131.33(ft) [11] 113.96(ft) [12] 179.08(ft)

[BOM] User-defined Path =====

1.
start = start
end = end
SpaceStart = [37] Public Lobby
SpaceEnd = [78] CEO Office
distance = 100.48
depth = 4
numberOfTurn = 7
PathDef = bera.bom.PathDef@de6cfc
id = 1
name = myPath
type = Path
area = 3696.47
height = 11.0
volume = 39780.26
numberOfSpace = 6
numberOfFloor = 0
Space = [51] Front Desk > [57] Business Hall > [46] Corridor > [76]
Secretary >

2.
start = start
end = end
SpaceStart = [37] Public Lobby
SpaceEnd = [28] Cooperation Office
distance = 66.39
depth = 3
numberOfTurn = 4
PathDef = bera.bom.PathDef@de6cfc
id = 2
name = myPath
type = Path
area = 2411.43
height = 10.0

```

```

volume = 24114.3
numberOfSpace = 5
numberOfFloor = 0
Space = [49] Rest Room > [45] Shared Office Support > [46] Corridor >

...

12.
start = start
end = end
SpaceStart = [96] Level 2 Lobby
SpaceEnd = [71] Visitors Office
distance = 179.08
depth = 8
numberOfTurn = 11
PathDef = bera.bom.PathDef@de6cfc
id = 12
name = myPath
type = Path
area = 5665.11
height = 13.25
volume = 62876.86
numberOfSpace = 10
numberOfFloor = 0
Space = [97] Elevator > [38] Elevator > [38] Elevator > [37] Public
Lobby > [51] Front Desk > [57] Business Hall > [59] Restricted Corridor
> [46] Corridor >

```

6.4. Application for Evaluating Spatial Program

6.4.1. SpaceGroup Rule

The former section described how static and dynamic building objects could be instantiated and handled by users. This section shows how to use them: actual applications for evaluating spatial program rules. Table 6.6 depicts a spatial program rule named ‘officeRule’ that is defining some rule conditions. For being a reusable rule, its arguments can pass a SpaceGroup object named ‘input’. Also there is another rule named ‘officeRule2’ which has three more rule conditions. It is inherited from ‘officeRule’,

therefore it has total 6 rule conditions. A dynamically instantiated SpaceGroup object²³ named ‘research’ is consumed by the rule, and the rule execution result is displayed in Table 6.6. Of course, the results from the rule execution from ‘officeRule’ and ‘officeRule2’ are different as the BERA console displays. Only one space object satisfies total 6 rule conditions from ‘officeRule2’, among three space objects that are passed in ‘officeRule’. This example also demonstrates an IF ~ ELSE ~ statement in its rule execution. For the test purpose, Table 6.6 displays all the results. (Only ‘officeRule2’ will be executed and displays the result because ‘officeRule’ is fail in this example)

Table 6.6. To evaluate a rule named ‘officeRule’ using input space group named ‘research’. The entire set of ‘research’ is displayed in the BERA console as below, and which ones are passed or not also printed out in the console area. Three passed space objects are highlighted in 3D visualization. But only one space object is passed in the rule checking for ‘officeRule2’ that is inherited from ‘officeRule’.

```
Space research = getSpace(Space.department = "research");

Rule officeRule(Space input) {
    input.name = "office";
    input.area > 200;
    input.height > 9;
}

Rule officeRule2(Space input) extends officeRule {
    input.numberOfWindow >= 1;
    input.Floor.height > 12;
}
```

²³ This example shows a specific spatial classification named “department”, and it could be derived from the model via IfcZone, IfcRelClassification, or a customized IfcProperty entity within the IFC scheme. The test model’s spaces have their departmental names defined in a Revit-specific property set named “Department”. How to define and retrieve this kind of information, especially within the IFC, is an MVD related issue [MVD, 2010]. Also there are many open-discussions or standardization efforts on the spatial classification issues such as OmniClass [OCCS, 2010], GSA PBS [GSA, 2010b], etc [Lee J-K et al, 2010].

```

        input.Floor.one.Space.name = "conference";
    }

    get(research);

    if (officeRule(research)) officeRule(research)
    else officeRule2(research);

```



```

[BERA] Parsing bBOMDef =====
1.
name = research
BOMType = Space
condition = getSpace(Space.department = "research")
Definition BOM Prop:
lop =
operand_left = Space.department
operator = =
operand_right = "research"
- BOM1 = Space
- property = department

[BERA] Parsing bRuleDef =====
1.
name = officeRule
args = Space input
ruleType = SpaceGroup
condition = input.name = "office";
            input.area > 200;
            input.height > 9;
Definition Rule's BOM Prop:

```

```

lop =
operand_left = input.name
operator = =
operand_right = "office"
- container = input
- property = name
lop =
operand_left = input.area
operator = >
operand_right = 200
- container = input
- property = area
lop =
operand_left = input.height
operator = >
operand_right = 9
- container = input
- property = height
2.
name = officeRule2
args = Space input
ruleType = SpaceGroup
inheritedFrom = officeRule
condition = input.name = "office";
            input.area > 200;
            input.height > 9;
input.numberOfWindow >= 1;
            input.Floor.height > 12;
            input.Floor.one.Space.name = "conference";
Definition Rule's BOM Prop:
lop =
operand_left = input.name
operator = =
operand_right = "office"
- container = input
- property = name
lop =
operand_left = input.area
operator = >
operand_right = 200
- container = input
- property = area
lop =
operand_left = input.height
operator = >
operand_right = 9
- container = input
- property = height
lop =
operand_left = input.numberOfWindow
operator = >=
operand_right = 1
- container = input
- property = numberOfWindow
lop =
operand_left = input.Floor.height
operator = >

```

```

operand_right = 12
- container = input
- BOM1 = Floor
- property = height
lop =
operand_left = input.Floor.one.Space.name
operator = =
operand_right = "conference"
- container = input
- BOM1 = Floor
- quant2 = one
- BOM2 = Space
- property = name

[BERA] Parsing bExeStat =====
1.
exeCommand = get
args = research
argsType1 = BID
argsType2 =
2.
exeCommand = officeRule
args = research
argsType1 = BID
argsType2 =
3.
exeCommand = officeRule2
args = research
argsType1 = BID
argsType2 =

[BOM] User-defined SpaceGroup =====

1.
id = 1
name = research
type = Space
area = 9908.51
height = 10.22
volume = 103446.8
windowArea = 396.0
numberOfSpace = 23
numberOfWindow = 22
Space = [93] TELE [35] MEN Restroom [36] WOMEN Restroom [32] Public
Team [33] Press Room [30] Demo Room [31] PR officer [13] Break Area
[14] Office [10] Research Team 3 [11] Restricted Corridor [12]
Development Office [8] Research Team 2 [9] Research Office 3 [5]
Research Office 1 [6] Research Team 1 [7] Research Office 2 [2]
Software Team [4] Laboratory [1] Head Researcher [91] Conference Room
[92] Research Chamber [89] Server Room

[BERA] Language Execution # 1 =====
command = get, args = research

[BERA] Language Execution # 2 =====
command = officeRule, args = research

```

```

passFail = fail

passObject =
--- [31] PR officer
--- [14] Office
--- [12] Development Office

failObject =
--- [93] TELE
--- [35] MEN Restroom
--- [36] WOMEN Restroom
--- [32] Public Team
--- [33] Press Room
--- [30] Demo Room
--- [13] Break Area
--- [10] Research Team 3
--- [11] Restricted Corridor
--- [8] Research Team 2
--- [9] Research Office 3
--- [5] Research Office 1
--- [6] Research Team 1
--- [7] Research Office 2
--- [2] Software Team
--- [4] Laboratory
--- [1] Head Researcher
--- [91] Conference Room
--- [92] Research Chamber
--- [89] Server Room

[BERA] Language Execution # 3 =====
command = officeRule2, args = research

passFail = fail

passObject =
--- [31] PR officer

failObject =
--- [93] TELE
--- [35] MEN Restroom
--- [36] WOMEN Restroom
--- [32] Public Team
--- [33] Press Room
--- [30] Demo Room
--- [13] Break Area
--- [14] Office
--- [10] Research Team 3
--- [11] Restricted Corridor
--- [12] Development Office
--- [8] Research Team 2
--- [9] Research Office 3
--- [5] Research Office 1
--- [6] Research Team 1
--- [7] Research Office 2
--- [2] Software Team
--- [4] Laboratory
--- [1] Head Researcher

```

```
--- [91] Conference Room
--- [92] Research Chamber
--- [89] Server Room
```

6.4.2. Floor Rule

As shown in the former SpaceGroup rule section, almost unlimited numbers of spatial rules can be created and checked by users based on the Space and SpaceGroup objects and their properties as many as the BOM defines. A floor is technically another collection of spaces which have same elevation height; therefore the BERA Language Tool handles it as another type of dynamic BOM where it has been instantiated by users. This section shows another type of rule using floor objects as Table 6.7 describes.

Table 6.7. To instantiate two floor objects named ‘floor1’ and ‘floor2’, and put those into a user-defined rule named ‘floorRule’ which regulates certain floor conditions. This example evaluates two checking instances using two input floor parameters. One is passed and another is failed as described in this table.

```
Floor floor1 = getFloor("Level 1");
Floor floor2 {
    Floor.number < 0;
}

Rule floorRule(Floor f) {
    f.area > 1000;
    f.height > 10;
    f.name = "level";
    f.Space.height > 8;
}

floorRule(floor1);
floorRule(floor2);
get(floor1);
get(floor2);
```



```
[BERA] Parsing bBOMDef =====
1.
name = floor1
BOMType = Floor
condition = getFloor("Level 1")
Definition BOM Prop:
lop =
operand_left = Space.name
operator = =
operand_right = "Level 1"
- BOM1 = Space
- property = name
2.
name = floor2
BOMType = Floor
condition = Floor.number < 0;
Definition BOM Prop:
lop =
operand_left = Floor.number
operator = <
operand_right = 0
- BOM1 = Floor
- property = number

[BERA] Parsing bRuleDef =====
1.
name = floorRule
args = Floor f
ruleType = Floor
condition = f.area > 1000;
```

```

        f.height > 10;
        f.name = "level";
        f.Space.height > 8;
Definition Rule's BOM Prop:
lop =
operand_left = f.area
operator = >
operand_right = 1000
- container = f
- property = area
lop =
operand_left = f.height
operator = >
operand_right = 10
- container = f
- property = height
lop =
operand_left = f.name
operator = =
operand_right = "level"
- container = f
- property = name
lop =
operand_left = f.Space.height
operator = >
operand_right = 8
- container = f
- BOM1 = Space
- property = height

[BERA] Parsing bExeStat =====
1.
exeCommand = floorRule
args = floor1
argsType1 = BID
argsType2 =
2.
exeCommand = floorRule
args = floor2
argsType1 = BID
argsType2 =
3.
exeCommand = get
args = floor1
argsType1 = BID
argsType2 =
4.
exeCommand = get
args = floor2
argsType1 = BID
argsType2 =

[BOM] User-defined SpaceGroup =====

1.
id = 1
name = floor1

```

```

type = Floor
area = 40653.88
height = 20.0
volume = 813077.6
numberOfSpace = 0
numberOfFloor = 0
Floor = [bera.bom.Floor@1e45d17]

2.
id = 2
name = floor2
type = Floor
area = 2178.8
height = 16.0
volume = 34860.8
numberOfSpace = 0
numberOfFloor = 0
Floor = [bera.bom.Floor@19b9c3b]

[BERA] Language Execution # 1 =====
command = floorRule, args = floor1

passFail = pass
whatClass = class bera.bom.Floor

passObject =

--- [2] Level 1 (area = 40653.88 height = 20.0 elevationHeight = 0.0 )

failObject =

[BERA] Language Execution # 2 =====
command = floorRule, args = floor2

passFail = fail
whatClass = class bera.bom.Floor

passObject =

failObject =

--- [1] Basement (area = 2178.8 height = 16.0 elevationHeight = -16.0 )

[BERA] Language Execution # 3 =====
command = get, args = floor1

[BERA] Language Execution # 4 =====
command = get, args = floor2

```

6.5. Application for Evaluating Building Circulation

6.5.1. Circulation Rule

A circulation rule can be instantiated by the user based on two different spaces or space groups. The example described in Table 6.8 shows the two given parameters pre-defined dynamic BOM – SpaceGroup. They are always zero or many, thus the circulation path instances will be populated in actual execution stage. In this case, there are two start space instances and seven end spaces; therefore, total 14 circulation path instances are instantiated and evaluated. The result indicates that only four paths are passed, that is, only four paths are satisfied with given rule conditions defined by the user.

Table 6.8. To instantiate a circulation rule named ‘myOfficeCircRule’ and evaluate it using two pre-defined BOM named ‘visitorOffices’ and ‘meetingSpaces’. The execution statement is simply its rule name with two arguments as follows. All 14 path instances are visualized in SMC, and the BERA console displays their rule checking result in detail.

```
Space visitorOffices = getSpace("visitor", "office");
Space meetingSpaces = getSpace("meeting") + getSpace("conference") +
getSpace("lecture");

Rule myOfficeCircRule(Space start, Space end) {
    Path p = getPath(start, end);
    p.Space.security = "public";
    p.distance < 150;
    p.depth < 6;
}

myOfficeCircRule(visitorOffices, meetingSpaces);
```



```
[BERA] Parsing bBOMDef =====
1.
name = visitorOffices
BOMType = Space
condition = getSpace("visitor", "office")
Definition BOM Prop:
lop =
operand_left = Space.name
operator = =
operand_right = "visitor"
- BOM1 = Space
- property = name
lop =
operand_left = Space.name
operator = =
operand_right = "office"
- BOM1 = Space
- property = name
2.
name = meetingSpaces
BOMType = Space
condition = getSpace("meeting") + getSpace("conference") +
getSpace("lecture")
Definition BOM Prop:
lop =
operand_left = Space.name
operator = =
operand_right = "meeting"
- BOM1 = Space
- property = name
lop =
```

```

operand_left = Space.name
operator = =
operand_right = "conference"
  - BOM1 = Space
  - property = name
lop =
operand_left = Space.name
operator = =
operand_right = "lecture"
  - BOM1 = Space
  - property = name

[BERA] Parsing bRuleDef =====
1.
name = myOfficeCircRule
args = Space start, Space end
ruleType = Path
condition = Path p = getPath(start, end);
          p.Space.security = "public";
          p.distance < 150;
          p.depth < 6;
Definition Rule's BOM Prop:
lop =
operand_left = p.Space.security
operator = =
operand_right = "public"
  - container = p
  - BOM1 = Space
  - property = security
lop =
operand_left = p.distance
operator = <
operand_right = 150
  - container = p
  - property = distance
lop =
operand_left = p.depth
operator = <
operand_right = 6
  - container = p
  - property = depth
1-1 Nested BOM:
    name = p
    BOMType = Path
    condition = getPath(start, end)
lop =
    operand_left = start
    operator = Path
    operand_right = end

[BERA] Parsing bExeStat =====
1.
exeCommand = myOfficeCircRule
args = visitorOffices, meetingSpaces
argsType1 = BID
argsType2 = BID
[BERA] No circulation start space found.

```

```

[BERA] No circulation end space found.

[BOM] User-defined SpaceGroup =====

1.
id = 1
name = visitorOffices
type = Space
area = 1085.91
height = 11.0
volume = 11779.76
numberOfSpace = 2
Space = [52] Visitors Office [71] Visitors Office

2.
id = 2
name = meetingSpaces
type = Space
area = 2854.48
height = 11.0
volume = 31785.39
numberOfSpace = 7
Space = [34] Visitors Conference [18] Lecturers Office [44] Conference
Room [24] Lecture Room [25] Conference [19] Small Lecture Room [91]
Conference Room

[BERA] Language Execution # 1 =====
command = myOfficeCircRule, args = visitorOffices, meetingSpaces

[BERA] Path definition in a Rule:myOfficeCircRule

passFail = fail

passObject =

--- [5] 141.71(ft) : Visitors Conference > Business Hall > Front Desk >
Public Lobby > Academic Hall >
--- [6] 131.05(ft) : Visitors Conference > Business Hall > Front Desk >
Public Lobby > Academic Hall >
--- [8] 106.76(ft) : Corridor > Restricted Corridor > Business Hall >
--- [12] 130.53(ft) : Corridor > Academic Hall >

failObject =

--- [1] 0.0(ft) :
--- [2] 123.46(ft) : Visitors Conference > Business Hall > Front Desk >
Public Lobby > Hall >
--- [3] 118.6(ft) : Visitors Conference > Business Hall > Front Desk >
Public Lobby > Rest Room > Shared Office Support > Corridor >
--- [4] 166.06(ft) : Visitors Conference > Business Hall > Front Desk >
Public Lobby > Academic Hall > Multimedia Open Library >
--- [7] 87.78(ft) : Visitors Conference > Restricted Corridor >
--- [9] 193.77(ft) : Corridor > Academic Hall > Small Lecture Room >
--- [10] 104.56(ft) : Corridor >
--- [11] 167.18(ft) : Corridor > Academic Hall > Multimedia Open
Library >

```

```
--- [13] 172.99(ft) : Corridor > Academic Hall >
--- [14] 198.05(ft) : Corridor > Restricted Corridor > Business Hall >
Visitors Conference > Restricted Corridor >
```

```
[BOM] User-defined PathDef =====
```

```
3.
id = 3
name = p
start = start
end = end
numberOfStartSpace = 2
numberOfEndSpace = 7
numberOfPaths = 14
startSpace = [52] Visitors Office [71] Visitors Office
endSpace = [34] Visitors Conference [18] Lecturers Office [44]
Conference Room [24] Lecture Room [25] Conference [19] Small Lecture
Room [91] Conference Room
Path = [1] 0.0(ft) [2] 123.46(ft) [3] 118.6(ft) [4] 166.06(ft) [5]
141.71(ft) [6] 131.05(ft) [7] 87.78(ft) [8] 106.76(ft) [9] 193.77(ft)
[10] 104.56(ft) [11] 167.18(ft) [12] 130.53(ft) [13] 172.99(ft) [14]
198.05(ft)
```

```
[BOM] User-defined Path =====
```

```
1.
start = start
end = end
SpaceStart = [52] Visitors Office
SpaceEnd = [34] Visitors Conference
distance = 0.0
depth = 0
numberOfTurn = 0
PathDef = bera.bom.PathDef@fed540
id = 1
name = p
type = Path
area = 0.0
height = 0.0
volume = 0.0
numberOfSpace = 2
numberOfFloor = 0
Space =
```

```
2.
start = start
end = end
SpaceStart = [52] Visitors Office
SpaceEnd = [18] Lecturers Office
distance = 123.46
depth = 5
numberOfTurn = 7
PathDef = bera.bom.PathDef@fed540
id = 2
name = p
type = Path
```

```

area = 3847.55
height = 11.6
volume = 44515.0
numberOfSpace = 7
numberOfFloor = 0
Space = [34] Visitors Conference > [57] Business Hall > [51] Front Desk
> [37] Public Lobby > [17] Hall >

...

14.
start = start
end = end
SpaceStart = [71] Visitors Office
SpaceEnd = [91] Conference Room
distance = 198.05
depth = 5
numberOfTurn = 9
PathDef = bera.bom.PathDef@fed540
id = 14
name = p
type = Path
area = 6337.12
height = 10.8
volume = 67118.18
numberOfSpace = 7
numberOfFloor = 0
Space = [46] Corridor > [59] Restricted Corridor > [57] Business Hall >
[34] Visitors Conference > [11] Restricted Corridor >

```

6.5.2. BERA and Target Language-based Execution

The example in Table 6.9 describes an advanced application and evaluation using the BERA Language program with its target language – in this implementation, it is Java. Table 6.9 shows a circulation rule checking BERA program and an advanced use of Java program code. The BERA Language User Manual in Appendix D describes the detailed features for this advanced mode. The Java code in this example aims to figure out what is the most remote space from “visitors conference”, what path instance has the most number of turns from the same space, and what path instance has the most number of

intermediate spaces (spatial depth) from it. The result emits this advanced result: 1) “[22] Rest room” is the most remote space from it, 2) 10 turns required to reach “[27] Human Resources Office” from it, and 3) seven spatial depths to “[27] Human Resources Office” is the third result.

Table 6.9. To instantiate circulation paths from “visitors conference” to all the spaces in the same floor – in this example, 92 path instances are populated. Not only is its rule checking result, but also an advanced Java program result also displayed in the BERA console.

```

Space allGroundSpaces = getSpace(Space.Floor.number = 1);
Rule distanceRule(Space start, Space target) {
    Path path = getPath(start, target);
    path.distance < 100;
    path.depth < 5;
}

distanceRule("visitors conference", allGroundSpaces);

java;

Double maxDistance = 0.0;
int maxTurn = 0;
int maxDepth = 0;
Space end1 = null;
Space end2 = null;
Space end3 = null;

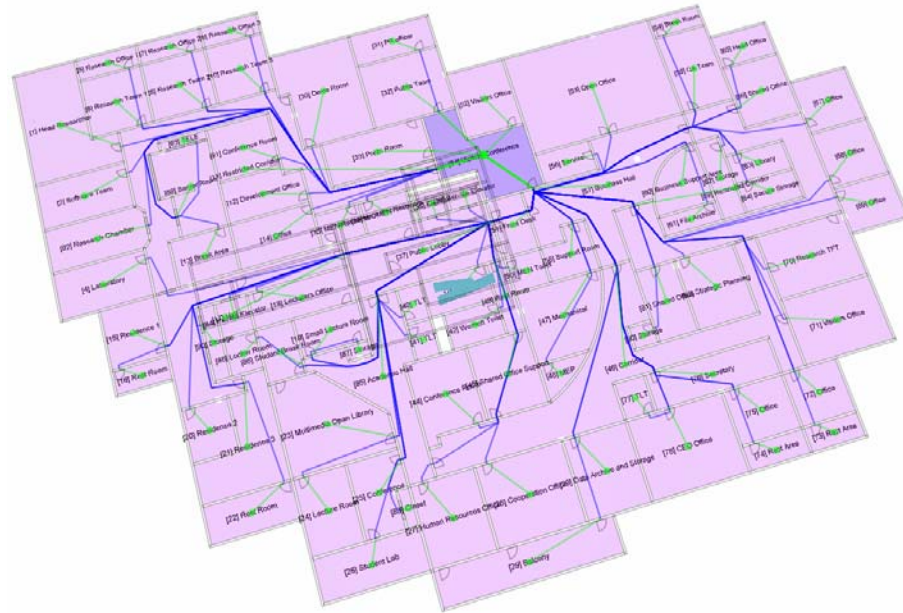
for (Path p : Path) {
    if (p.distance >= maxDistance) {
        maxDistance = p.distance;
        end1 = p.SpaceEnd;
    }
    if (p.numberOfTurn >= maxTurn) {
        maxTurn = p.numberOfTurn;
        end2 = p.SpaceEnd;
    }
    if (p.depth >= maxDepth) {
        maxDepth = p.depth;
        end3 = p.SpaceEnd;
    }
}

String maxDistanceSpace = "[" + end1.number + "]" + end1.name;
String maxTurnSpace = "[" + end2.number + "]" + end2.name;

```

```
String maxDepthSpace = "[" + end3.number + "]" + end3.name;

print("MAX distance from \"visitors conference\" to" + maxDistanceSpace
+ " = " + maxDistance);
print("MAX num of turns from \"visitors conference\" to " +
maxTurnSpace + " = " + maxTurn);
print("MAX depth from \"visitors conference\" to " + maxDepthSpace + "
= " + maxDepth);
```



```
[BERA] Parsing bBOMDef =====
1.
name = allGroundSpaces
BOMType = Space
condition = getSpace(Space.Floor.number = 1)
Definition BOM Prop:
lop =
operand_left = Space.Floor.number
operator = =
operand_right = 1
- BOM1 = Space
- BOM2 = Floor
- property = number

[BERA] Parsing bRuleDef =====
1.
name = distanceRule
args = Space start, Space target
ruleType = Path
condition = Path path = getPath(start, target);
```

```

        path.distance < 100;
        path.depth < 5;
Definition Rule's BOM Prop:
lop =
operand_left = path.distance
operator = <
operand_right = 100
- container = path
- property = distance
lop =
operand_left = path.depth
operator = <
operand_right = 5
- container = path
- property = depth
1-1 Nested BOM:
    name = path
    BOMType = Path
    condition = getPath(start, target)
lop =
    operand_left = start
    operator = Path
    operand_right = target

[BERA] Parsing bExeStat =====
1.
exeCommand = distanceRule
args = "visitors conference", allGroundSpaces
argsType1 = bStringQuot
argsType2 = BID
args 1
    name = noname_1
    BOMType = Space
    condition = getSpace("visitors conference")
    Definition BOM Prop:
lop =
    operand_left = Space.name
    operator = =
    operand_right = "visitors conference"
- BOM1 = Space
- property = name
[BERA] No circulation start space found.
[BERA] No circulation end space found.

[BOM] User-defined SpaceGroup =====
1.
id = 1
name = allGroundSpaces
type = Space
area = 37492.56
height = 10.78
volume = 404904.39
numberOfSpace = 92
numberOfFloor = 0
Space = [90] Storage [83] Closet [93] TELE [38] Elevator [39] Service
Elevator [35] MEN Restroom [36] WOMEN Restroom [32] Public Team [33]

```

Press Room [34] Visitors Conference [29] Balcony [30] Demo Room [31] PR officer [18] Lecturers Office [48] MEP [16] Rest Room [49] Rest Room [17] Hall [50] MEN Toilet [45] Shared Office Support [13] Break Area [46] Corridor [14] Office [47] Mechanical [15] Residence 1 [10] Research Team 3 [43] Stair [11] Restricted Corridor [44] Conference Room [81] Shared Office [12] Development Office [82] Strategic Planning [77] TLT [40] TLT [27] Human Resources Office [78] CEO Office [41] TLT [28] Cooperation Office [79] Data Archive and Storage [42] Women Toilet [24] Lecture Room [75] Office [25] Conference [76] Secretary [26] Student Lab [21] Residence 3 [72] Office [37] Public Lobby [22] Rest Room [80] Storage [59] Restricted Corridor [73] Rest Area [60] Business Support Area [23] Multimedia Open Library [74] Rest Area [56] Service [19] Small Lecture Room [57] Business Hall [20] Residence 2 [58] Support Room [53] Open Office [54] Break Room [55] QA Team [51] Front Desk [52] Visitors Office [69] Office [70] Research TFT [71] Visitors Office [67] Office [68] Office [64] Secure Storage [65] Head Office [66] Shared Office [61] File Archive [62] Storage [63] Library [8] Research Team 2 [9] Research Office 3 [5] Research Office 1 [6] Research Team 1 [7] Research Office 2 [2] Software Team [4] Laboratory [1] Head Researcher [91] Conference Room [92] Research Chamber [89] Server Room [86] Student Break Room [85] Academic Hall [84] Private Elevator [87] Storage [88] Locker Room

```

3.
id = 3
name = noname_1
type = Space
area = 684.67
height = 12.0
volume = 8216.04
numberOfSpace = 1
numberOfFloor = 0
Space = [34] Visitors Conference

```

```

[BERA] Language Execution # 1 =====
command = distanceRule, args = "visitors conference", allGroundSpaces

```

```

[BERA] Path definition in a Rule:distanceRule

```

```

passFail = fail

```

```

passObject =

```

```

--- [4] 40.23(ft) : Business Hall > Front Desk > Public Lobby >
--- [5] 30.92(ft) : Business Hall > Front Desk > Public Lobby >
--- [6] 32.45(ft) : Restricted Corridor >
--- [7] 26.48(ft) : Restricted Corridor >
--- [8] 0.0(ft) :
--- [9] 0.0(ft) :
--- [10] 0.0(ft) :
--- [12] 53.21(ft) : Restricted Corridor >
--- [13] 23.5(ft) : Public Team >
--- [15] 68.25(ft) : Business Hall > Support Room > Mechanical >
--- [17] 35.41(ft) : Business Hall > Front Desk > Public Lobby >
--- [18] 69.6(ft) : Business Hall > Front Desk > Public Lobby >
--- [19] 40.54(ft) : Business Hall > Front Desk > Public Lobby > Rest

```

```

Room >
--- [20] 59.38(ft) : Business Hall > Front Desk > Public Lobby > Rest
Room >
--- [22] 38.78(ft) : Business Hall >
--- [23] 43.04(ft) : Restricted Corridor >
--- [24] 34.81(ft) : Business Hall > Support Room >
--- [26] 78.44(ft) : Restricted Corridor >
--- [27] 37.38(ft) : Business Hall > Front Desk > Public Lobby >
--- [28] 0.0(ft) :
--- [30] 55.89(ft) : Business Hall > Restricted Corridor >
--- [31] 43.26(ft) : Restricted Corridor >
--- [32] 58.65(ft) : Business Hall > Restricted Corridor >
--- [33] 98.28(ft) : Business Hall > Corridor > Secretary > CEO Office
>
--- [34] 77.95(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall >
--- [36] 89.14(ft) : Business Hall > Corridor > Secretary >
--- [37] 83.35(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall >
--- [39] 95.64(ft) : Business Hall > Corridor >
--- [40] 51.84(ft) : Business Hall > Front Desk > Public Lobby > Rest
Room >
--- [44] 77.13(ft) : Business Hall > Corridor >
--- [48] 20.92(ft) : Business Hall > Front Desk >
--- [50] 76.98(ft) : Business Hall > Restricted Corridor > Shared
Office >
--- [51] 34.73(ft) : Business Hall >
--- [53] 55.87(ft) : Business Hall >
--- [56] 33.48(ft) : Business Hall >
--- [58] 0.0(ft) :
--- [60] 17.64(ft) : Business Hall >
--- [61] 39.28(ft) : Business Hall >
--- [62] 94.27(ft) : Business Hall > Restricted Corridor > QA Team >
--- [63] 65.75(ft) : Business Hall > Restricted Corridor >
--- [64] 8.92(ft) : Business Hall >
--- [65] 0.0(ft) :
--- [67] 89.99(ft) : Business Hall > Restricted Corridor > Corridor >
--- [69] 97.77(ft) : Business Hall > Restricted Corridor >
--- [71] 87.99(ft) : Business Hall > Restricted Corridor > Library >
--- [72] 92.7(ft) : Business Hall > Restricted Corridor > Shared Office
>
--- [73] 76.28(ft) : Business Hall > Restricted Corridor >
--- [74] 77.04(ft) : Business Hall > Business Support Area >
--- [75] 74.84(ft) : Business Hall > Restricted Corridor >
--- [76] 77.71(ft) : Business Hall > Restricted Corridor >
--- [77] 94.57(ft) : Restricted Corridor >
--- [78] 99.94(ft) : Restricted Corridor > Research Team 3 >
--- [85] 57.83(ft) : Restricted Corridor >
--- [89] 66.17(ft) : Business Hall > Front Desk > Public Lobby >

failObject =

--- [1] 128.46(ft) : Business Hall > Front Desk > Public Lobby > Hall >
--- [2] 137.85(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall > Conference >
--- [3] 173.34(ft) : Restricted Corridor > Server Room >
--- [11] 121.05(ft) : Business Hall > Corridor > Data Archive and

```

```
Storage >
--- [14] 100.96(ft) : Business Hall > Front Desk > Public Lobby > Hall
>
--- [16] 152.55(ft) : Business Hall > Front Desk > Public Lobby > Hall
> Residence 1 >
--- [21] 159.96(ft) : Restricted Corridor > Server Room >
--- [25] 135.63(ft) : Business Hall > Front Desk > Public Lobby > Hall
>
--- [29] 96.1(ft) : Business Hall > Front Desk > Public Lobby > Rest
Room > Shared Office Support > Corridor >
--- [35] 119.64(ft) : Business Hall > Front Desk > Public Lobby > Rest
Room > Shared Office Support > Corridor > Academic Hall >
--- [38] 101.8(ft) : Business Hall > Front Desk > Public Lobby > Rest
Room > Shared Office Support > Corridor >
--- [41] 143.56(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall > Multimedia Open Library >
--- [42] 103.49(ft) : Business Hall > Corridor > Secretary >
--- [43] 119.21(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall >
--- [45] 144.69(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall > Conference >
--- [46] 157.7(ft) : Business Hall > Front Desk > Public Lobby > Hall >
--- [47] 121.3(ft) : Business Hall > Restricted Corridor > Corridor >
--- [49] 191.94(ft) : Business Hall > Front Desk > Public Lobby > Hall
> Residence 3 >
--- [52] 138.71(ft) : Business Hall > Restricted Corridor > Corridor >
Office >
--- [54] 112.25(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall >
--- [55] 120.9(ft) : Business Hall > Corridor > Secretary > Office >
--- [57] 108.55(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall >
--- [59] 152.61(ft) : Business Hall > Front Desk > Public Lobby > Hall
>
--- [66] 109.45(ft) : Business Hall > Restricted Corridor >
--- [68] 106.76(ft) : Business Hall > Restricted Corridor > Corridor >
--- [70] 110.6(ft) : Business Hall > Restricted Corridor >
--- [79] 137.77(ft) : Restricted Corridor > Research Team 1 >
--- [80] 116.27(ft) : Restricted Corridor >
--- [81] 116.07(ft) : Restricted Corridor > Research Team 2 >
--- [82] 126.18(ft) : Restricted Corridor >
--- [83] 142.61(ft) : Business Hall > Front Desk > Public Lobby > Hall
> Restricted Corridor >
--- [84] 122.68(ft) : Restricted Corridor >
--- [86] 143.79(ft) : Restricted Corridor >
--- [87] 144.76(ft) : Restricted Corridor >
--- [88] 109.64(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall >
--- [90] 109.72(ft) : Business Hall > Front Desk > Public Lobby > Hall
>
--- [91] 126.1(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall > Small Lecture Room >
--- [92] 128.27(ft) : Business Hall > Front Desk > Public Lobby >
Academic Hall > Student Break Room >
```

```
[BOM] User-defined PathDef =====
```

```

2.
id = 2
name = path
start = start
end = target
numberOfStartSpace = 1
numberOfEndSpace = 92
numberOfPaths = 92
startSpace = [34] Visitors Conference
endSpace = [90] Storage [83] Closet [... [88] Locker Room
Path = [1] 128.46(ft) [2] 137.85(ft) ... [92] 128.27(ft)

[BOM] User-defined Path =====

1.
start = start
end = target
SpaceStart = [34] Visitors Conference
SpaceEnd = [90] Storage
distance = 128.46
depth = 4
numberOfTurn = 6
id = 1
name = path
type = Path
area = 3162.88
height = 11.5
volume = 36298.96
numberOfSpace = 6
Space = [57] Business Hall > [51] Front Desk > [37] Public Lobby > [17]
Hall >

2.
start = start
end = target
SpaceStart = [34] Visitors Conference
SpaceEnd = [83] Closet
distance = 137.85
depth = 5
numberOfTurn = 8
id = 2
name = path
type = Path
area = 3456.37
height = 12.2
volume = 43214.86
numberOfSpace = 7
Space = [57] Business Hall > [51] Front Desk > [37] Public Lobby > [85]
Academic Hall > [25] Conference >

...

92.
start = start
end = target
SpaceStart = [34] Visitors Conference
SpaceEnd = [88] Locker Room

```

```
distance = 128.27
depth = 5
numberOfTurn = 9
id = 92
name = path
type = Path
area = 3327.43
height = 12.2
volume = 41925.46
numberOfSpace = 7
Space = [57] Business Hall > [51] Front Desk > [37] Public Lobby > [85]
Academic Hall > [86] Student Break Room >
```

```
[BERA] Target executor file created successfully: BExecutor.java
[BERA] Runtime compilation succeeded: BExecutor.class
```

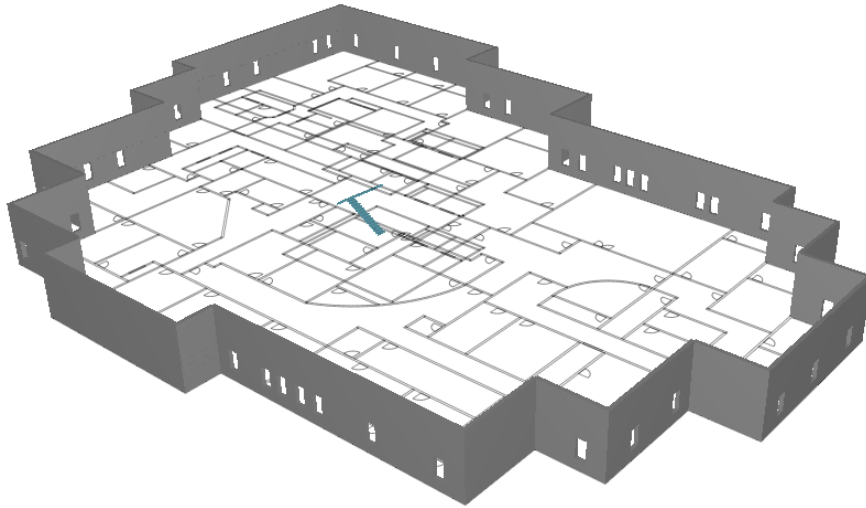
```
MAX distance from "visitors conference" to[22] Rest Room = 191.94
MAX num of turns from "visitors conference" to [27] Human Resources
Office = 10
MAX depth from "visitors conference" to [27] Human Resources Office = 7
```

6.6. Application for other Building Objects

As an additional example of non-spatial object and its application, this section introduces an application for evaluating wall objects. Syntactic definition for the wall object is already defined in the language definition chapter. In the scope of work, the properties for the Wall object has not yet been developed in detail, however, by using its derived properties from the building model such as GUID, type, area, volume, height, length, thickness, number of openings, and relational objects, a test case can be demonstrated as shown in Table 6.10. It evaluates given wall objects' thickness whether it is same or thicker than 1 foot or not, and the input object is a collection of walls that contain "exterior" in their type description. As a result, total 26 exterior wall objects are instantiated and evaluated the rule, and the rule checking result is "pass".

Table 6.10. To evaluate a simple test rule named 'externalWallRule' using input object group named 'exWalls' which contains Wall object instances. The associated information for 'my' is displayed in the BERA console as below, and which ones are passed or not also printed out in the console area. Three passed Wall objects are visualized in 3D.

```
Wall exWalls {  
    Wall.type = "exterior";  
}  
  
Rule externalWallRule(Wall wa) {  
    wa.thickness >= 1.0;  
}  
  
externalWallRule(exWalls);
```



```
[BERA] Parsing bBOMDef =====  
1.  
name = exWalls  
BOMType = Wall  
condition = Wall.type = "exterior";  
Definition BOM Prop:  
lop =  
operand_left = Wall.type  
operator = =  
operand_right = "exterior"  
- BOM1 = Wall
```

```

- property = type

[BERA] Parsing bRuleDef =====
1.
name = externalWallRule
args = Wall wa
ruleType = Structure
condition = wa.thickness >= 1.0;
Definition Rule's BOM Prop:
lop =
operand_left = wa.thickness
operator = >=
operand_right = 1.0
- container = wa
- property = thickness

[BERA] Parsing bExeStat =====
1.
exeCommand = externalWallRule
args = my
argsType1 = BID
argsType2 =

[BOM] User-defined Structure =====

1.
id = 1
name = exWalls
type = Wall
numberOfObject = 26
Property = {}
Slab = []
Column = []
Wall = [bera.bom.Wall@b6e46, ..., bera.bom.Wall@7a00b]
Door = []
Stair = []
Ramp = []
Window = []

[BERA] Runner initiated.

[BERA] Language Execution # 1 =====
command = externalWallRule, args = exWalls

passFail = pass

passObject =

--- [66] Basic Wall:Exterior - Brick and CMU on MTL. Stud:199977
--- [68] Basic Wall:Exterior - Brick and CMU on MTL. Stud:199853
--- [69] Basic Wall:Exterior - Brick and CMU on MTL. Stud:200010
--- [70] Basic Wall:Exterior - Brick and CMU on MTL. Stud:200026
--- [71] Basic Wall:Exterior - Brick and CMU on MTL. Stud:200030
--- [73] Basic Wall:Exterior - Brick and CMU on MTL. Stud:199727
--- [75] Basic Wall:Exterior - Brick and CMU on MTL. Stud:199900
--- [76] Basic Wall:Exterior - Brick and CMU on MTL. Stud:199901
--- [78] Basic Wall:Exterior - Brick and CMU on MTL. Stud:199801

```

```
--- [112] Basic Wall:Exterior - Brick and CMU on MTL. Stud:164023
--- [113] Basic Wall:Exterior - Brick and CMU on MTL. Stud:164068
--- [116] Basic Wall:Exterior - Brick and CMU on MTL. Stud:164137
--- [118] Basic Wall:Exterior - Brick and CMU on MTL. Stud:164156
--- [120] Basic Wall:Exterior - Brick and CMU on MTL. Stud:164110
--- [122] Basic Wall:Exterior - Brick and CMU on MTL. Stud:164265
--- [128] Basic Wall:Exterior - Brick and CMU on MTL. Stud:160939
--- [137] Basic Wall:Exterior - Brick and CMU on MTL. Stud:160739
--- [139] Basic Wall:Exterior - Brick and CMU on MTL. Stud:159808
--- [140] Basic Wall:Exterior - Brick and CMU on MTL. Stud:159838
--- [162] Basic Wall:Exterior - Brick and CMU on MTL. Stud:157175
--- [174] Basic Wall:Exterior - Brick and CMU on MTL. Stud:162590
--- [175] Basic Wall:Exterior - Brick and CMU on MTL. Stud:162649
--- [176] Basic Wall:Exterior - Brick and CMU on MTL. Stud:157260
--- [189] Basic Wall:Exterior - Brick and CMU on MTL. Stud:159624
--- [207] Basic Wall:Exterior - Brick and CMU on MTL. Stud:157836
--- [211] Basic Wall:Exterior - Brick and CMU on MTL. Stud:158084

failObject =
```

6.7. Evaluation of BERA Language

In this chapter, some actual BERA program applications are demonstrated regarding various issues can be dealt with the scope of the implementation, as well as an overview of extended building object: wall. The BERA Language Tool is not a magic wand, but this chapter demonstrates that it enables users to analyze building models in much easier than relying on general purposed programming languages (See the section 5.6.1 as an example comparison) and more powerful than pre-defined application interfaces such as table-based input parameters that are conventionally used in the rule checking software (Refer to the section 6.2.1 to compare them). As the BERA Language aims to be an easy and effective domain-specific language, the example programs in this chapter showed the fidelity to the actual building design rules and analysis issues even if they are some snippets of broader range of plausible BERA applications. As mentioned in

the introduction chapter, research questions and their answers can be summarized as follows (See the section 1.1.4 for the research questions):

- 1) Ease of use for BERA users: Based on the survey on the recently developed and popular languages, BERA takes advantages of dot-notation based access to the complex building objects and their relations.
- 2) BERA Object Model (BOM): BOM is the BERA user-centered abstraction of the target objects.
- 3) BOM and its dot-notation based access enables users to take advantages of explicit definition of target objects with ease. (See the section 3.2.4)
- 4) Handling real-world rules: Higher level design rules can be decomposed into the series of BERA notations within its domain-specific features. (See the section 3.2.2 and examples in this chapter)
- 5) Implementation issues: Chapter 5 demonstrated how the author implements the BERA Language and its Tool. Also its extensibility issues are described in the section 5.6 regarding both front-end and back-end extensibility.

The ultimate testing will come from the BERA Language users. The evaluation of the BERA Language and its Tool is one of the future works. For gaining user feedback and further updates, the open-ended testing and support arrangement is planned. Even if the assessment on the language is beyond the scope of this dissertation per se, the author has a plan to support live documents and feedback system through the web such as user manual, examples, demo videos, etc for better future BERA Language and its Tool. (Refer to the Appendix D, BERA User Manual, and BERA website for variety of applications and up-to-date demonstrations of BERA language could not be addressed in this dissertation.)

CHAPTER 7

CONCLUSION

The building model and its objects generated by BIM-authoring tools are smart and reusable to support subsequent-phase tasks such as design review, analysis and rule checking. These tasks impact the entire lifecycle of the building project and the quality of design. That is why one of the most recent and promising directions of BIM involves these tasks [Eastman et al, 2008]. However, existing applications for such tasks always have limitations because their capabilities, user interfaces and scopes are mostly pre-defined by programmers. This usually results in the development of another application or add-on software on top of existing applications. The research and development of the BERA Language described in this dissertation attempts to set users free from such limitations, multiple levels of application interface, or learning sophisticated general-purpose programming languages. One of the fundamental concepts of the BERA Language is to provide user-driven methods rather than software-driven methods on the design review tasks. A more important fact is that the language users mentioned here are not programmers but designers, architects, reviewers and others who are interested in building design review, analysis and rule checking. The BERA Language is meant to satisfy fundamental requirements for providing a ‘good’ domain-specific language: high fidelity to the problems with ease.

This research proposed the BERA Object Model (BOM): a human-centered abstraction of the complex state of real-world building models, rather than the computation-oriented abstraction which is generally intended to cover broad-ranged issues. BOM is one of the key concepts to the building environment rule and analysis as the language name literally implies. By using BOM, users can enjoy the ease of use and portability to the pre-defined BIM data, rather than sophisticated and platform-dependent

ways. A newly proposed BOM data structure has been implemented to cover spatial objects within the scope of this research and development which focuses on evaluating building circulation and spatial programming, but this dissertation also has described and demonstrated its open-endedness to cover the broader type of building object and its properties. The author realized that it is another challenge to define generic and valuable BOM as it grows more detailed. Both lateral extensions such as structural building elements and the vertical extensions such as additional properties for existing BOM objects are good examples of its open-endedness. In the BERA Language Tool implementation described in this dissertation, many computed and derived properties have been proposed and implemented for different purposes, as well as some basic data obtained directly from the given building model. These properties are available to users by dot-notations that are easy to read and write.

The BERA Language Tool is an integrated development environment for the proposed BERA Language. In its current incarnation, its BIM platform is SMC, and appears effective against complex building environment rule and analysis cases compared to existing software-driven methods. By using the BERA Language Tool, users can evaluate their rules on their given building models, focusing on both design analysis and rule checking purposes with respect to building circulation and spatial programming. The current rules and BOM were developed by the author as one of the possible directions to BERA regarding several different use cases. In Application and Evaluation chapter, it has been demonstrated the capabilities to support actual rules and analysis within the scope of this dissertation. The substantial benefits and potentials from using the BERA Language can be summarized as follows:

- 1) Ease of use: Contrary to general purpose languages, the BERA Language is easy to use for domain experts. It is almost equivalently effective as general purpose languages in the problem domain. (See the section 3.2.4 and 5.6.1 to compare the proposed BERA Language and general purpose languages)

- 2) Fidelity: The proposed BERA Language and its BOM structure have demonstrated its strength in high fidelity to the domain-specific issues. In this study, the building circulation and spatial programming issues are demonstrated. (See the chapter 6 for its various applications)
- 3) Extensibility: The BERA Language offers an explicit and extensible data model for the human-centered abstraction of a building – BOM. It is open-ended both laterally and vertically. (See the chapter 3 and section 5.6.2)
- 4) Portability: The proposed BERA Language is implemented on top of SMC as an actual development of the BERA Language Tool. However, BERA Language aims to be embedded in other various types of BIM-enabled applications such as BIM authoring tools, based on the platform-dependent back-end implementations. (See the section 5.6.1)

The BERA Language and its Tool has been carefully implemented and tested. The ultimate testing and evaluation of the language per se, however, will come from language users, as many users as possible. The open-ended testing, feedback system and support arrangement is planned for updated and upgraded BERA Language and its Tool (See Appendix D). The author expects that more development, especially on top of other types of building modeling platforms, have to be carried out by several entities including the author and his team so that there are constant contributions to the AEC industry and academia. Expected contributions of the development and use of BERA Language can be summarized as follows:

- 1) The BERA Language will allow for automated building design review and rule checking of BIM models to come into wide use.
- 2) The BERA Language is effective not only for the purpose of design rule checking, but also for various design analysis purposes. In other words, the rules in the BERA Language can be one of many possible user-defined rules even if they are

not relevant to existing real world rules. The BERA Language provides a massive analysis method for many building models in an efficient way.

- 3) This is the first attempt to develop a BIM domain-specific programming language focusing on building design issues. Therefore, we expect the BERA Language design and implementation to be a model for other domain-specific languages in other domains.
- 4) Within the scope of this study, the initial implementation focuses on spatial objects, group of spaces, circulation paths, their properties, and relations. This implementation will be a basic foundation that can be extended to other various building elements to cover other types of building environment rules and analysis as BERA literally implies.

Since the early efforts in the 1970's [Eastman, 1975; 1976; 1977], building information modeling (BIM) and its vast set of techniques have been developed by many researchers and developers in the area of design computation. BIM provides a solid foundation and the ultimate principle of the BERA Language. Efforts made by active participants and researchers in this domain have enabled the author to attempt to implement the BERA Language and its Tool. The author wishes to express his gratitude to those who have contributed to the associated area of research and development.

The BERA Language Tool implementation described in this dissertation is one of the outcomes of building environment rule and analysis – BERA. Development work has just begun: it is open-ended and still growing. The author believes that the proposed BERA Language and its tool development described in this dissertation have a positive and active influence upon current and future BIM-enabled applications in various disciplines.

APPENDIX A

EBNF NOTATION AS A CONTEXT-FREE GRAMMAR DEFINITION

EBNF (Extended Backus-Naur Form) is a meta-syntax notation for expressing context-free language grammars. It is used as a formal way to describe formal language syntaxes such as a computer programming language. EBNF is an extended form of BNF (Backus-Naur Form), and there are many variants. EBNF has been adopted as an international standard by ISO.

Following example shows how a lexical rule “**BID**” (BERA variable name identifier) can be represented in the EBNF notation. Also another simple rule “**bBIDQuot**” can be defined using multiple **BID**s with quotation marks.

```
BID           :    BIDprefix ( BIDprefix | BIDDigit )* ;
BIDprefix    :    'a'..'z'|'A'..'Z'|'_' ;
BIDDigit     :    '0'..'9' ;

bBIDQuot     :    '"' ( BID | BIDDigit )+ '"'
```

In actual programs, **BID** can represent any of following user-variable names without blanks: `mySpace`, `circulationRule2`, `MYRULE`, `program_3`, etc, but cannot support: `1234` or `3space`. The rule “**bBIDQuot**” can represent **BID**-based terminal strings including blanks, wrapped in double-quotation marks: “space name”, “Open office unit 3”, “property_ Values”, “N23 01”, etc. This is just a snippet of the EBNF notation, as used in the BERA Language grammar definition (See following appendices). For more details on EBNF/BNF notation, refer to [ISO/IEC 14977:1996, 2001; Johnson, 1979; Lesk, 1975] and many other references.

APPENDIX B

ANTLR AS A TOOL FOR DEFINING DOMAIN-SPECIFIC LANGUAGE

For developing the definition of BERA Language, ANTLR²⁴ (ANother Tool for Language Recognition) [Parr, 2010] has been used as a tool for defining this kind of domain-specific computer language. ANTLR was developed by Terence Parr [Parr, 2008; Parr, 2009], and one of the well-known tools used in developing the high-level domain-specific languages. In actual implementation of BERA Language Tool, ANTLR version 3.2 and ANTLR works version 1.4 have been used. ANTLR works is a GUI-based tool for developing a domain specific language.

In the world of advanced information technology, there are huge demands of developing domain-specific computer languages. In the earlier days when FORTRAN was the leading language, computer languages had been developed by Lex [Lesk, 1975] and Yacc [Johnson, 1979]. They are comparable tools with this BERA Language development using ANTLR. It has helped not only to define the BERA Language syntax, but also to implement the BERA Language Tool in some lexical and syntactic analysis stages for handling user-input text. The author wishes to express our gratitude to Terrence Parr for his efforts on developing this helpful parser generator for domain-specific languages. The entire BERA Language grammar definition is represented in the next appendix using EBNF notations that can be applicable to ANTLR works.

²⁴ ANTLR is a parser generator that is based on LL parsing and it is not restricted to finite tokens of look ahead. As one type of the context-free grammars, an LL parser is a top-down parser which parses the input text from left to right and constructs a leftmost derivation of the input sentence. For more detailed information: [Parr, 2010; www.antlr.org]

APPENDIX C

BERA LANGUAGE GRAMMAR

This appendix shows BERA Language grammar definition in the form of EBNF (a bit ANTLR-customized) as introduced in former appendices. The grammar definition is the actual code implemented in the BERA Language Tool version 1.0, and carefully tested. As other computer languages, this BERA grammar is also subject to be amended, elaborated and updated followed by additions of BERA Language. Some notations are ANTLR-specific (and ANTLR String Template [Parr, B, 2010]-specific) codes that are actually testable in the actual ANTLR environment. The definition here is one of the main subjects to be updated and fixed for the development of BERA Language Tool. (The definition described here is the latest version as of 2010 fall)

```
/* BERA Language Definition
 * Version - Bera_yyyymmdd - 20100808
 * For Building Environment Rule and Analysis (BERA) Language
Implementation
 * Jin-Kook Lee, leejinkook@gmail.com, Georgia Tech, CoA, Design
Computation
 * Advised by Professor Charles M. Eastman, Director, Digital
Building Laboratory
 * Copyright @ Georgia Institute of Technology, Design
Computation: Jin-kook Lee and Chuck Eastman, 2010.
 */
```

```
/* *****
```

```
BERA PARSER DEFINITION AREA
```

```

*****/

// *****

// BERA Program Body - BERABEGIN & BERAEND distinguishes BERA
Language & others e.g. Java

bProgram
    :    bBeraProgram EOF
    |    BERABEGIN ';' LT*
        bBeraProgram
        BERAEND ';' LT* EOF
    ;

// bBERAProgram is the main body & gate to the BERA Language:
Front-end Extensibility
bBeraProgram
    :
        bReference? bBOMDef? bRuleDef? bExeStat
    |    ';'
    ;

// *****

// Reference: External data for Space & Rule
bReference
    :    bBuildingTypeStat bRefStat*
    |    bRefStat+
    ;

bBuildingTypeStat

```

```

        :      BbuildingType
            ( BID | bStringQuot ) ';' LT*
        ;

bRefStat
    :      Breference bURL ';'
        |      Breference BID ( '.' BID)* ';'
    ;

bURL :      '"' bProtocol bURLadd '"'
    ;

bProtocol
    :      BProtocol
    ;

bURLadd
    :      BID ( BID | '/' | '.' | '%20' )*
    ;


// *****
// BOM Def - Space Definition & Getter
bBOMDef
    :      ( bBOMDecLines | bBOMDefStat )+
    ;

bBOMDecLines
    :      bBOMDecLine LT*
    ;

bBOMDecLine
    // e.g. Space officeSpace = getSpace("office");
    :      bWrapSpaceType bDecSingle ';'
        |      bWrapStructureType bDecSingle ';'
    ;

```

```

bDecSingle
// e.g. allRooms = getSpace("room");
      :      BID '=' bBOMGetter
      ;

bBOMGetter
// e.g. getPath("office", "lobby");
      :      (bGetterVerbs bBOMGetterP ( ('+'|'-') bBOMGetter ))
      |      bBOMGetterExpr
      ;

bBOMGetterP
      :      '('
              ( bMultiBOMGetter | bBOMGetterExpr )
              ')'
      ;

bGetterVerbs
      :      bBOMgetBOM
      |      bVerbs ( bWrapSpaceType | bWrapStructureType )
      ;

bMultiBOMGetter
      :      bStringRep | bStringQuotRep
      ;

bBOMgetBOM
      :      bBOMgetSpace
      |      bBOMgetStructure
      ;

bBOMgetSpace
      :      BgetBuilding
      |      BgetFloor
      |      BgetSpace
      |      BgetSpaceGroup

```

```

        |      BgetPath
    ;

bBOMgetStructure
    :      BgetStructure
        |      BgetSlab
        |      BgetColumn
        |      BgetWall
        |      BgetDoor
        |      BgetStair
        |      BgetRamp
    ;

bBOMDefStat
    :      bDefineBOM LT*
    ;

bDefineBOM
    :      bBOMDefStatDec bBOMDefBlock
    ;

bBOMDefStatDec
    :      Bdefine? (bWrapSpaceType | bWrapStructureType) BID LT*
    ;

bBOMDefBlock
    :      '{' LT* ( bBOMDeclines | bBOMPropExpr )+ '}'
    ;

bBOMPropExpr
    :      bBOMGetterExpr ';' LT*
    ;

bBOMGetterExpr
    :      bBOMGetterByBID

```

```

        |      bBOMGetterByProp
    ;

bBOMGetterByProp
//e.g. Space.Floor.name = "Level_1";
    :      bLogic?
          (bQuantifier '.')?
          ( bWrapSpaceType | bWrapStructureType )
          ( ( '.' BID ) | ( '.' (bQuantifier '.')? ( bWrapSpaceType
| bWrapStructureType ) ) | ( '.' BFunction) )+
          bComparisonOperator?
          (  BID  |  bStringQuot  |  '-'?  INTLITERAL  |  '-'?
DOUBLELITERAL )?
    ;

bBOMGetterByBID
//e.g. p.Space.area > 500.60;
    :      bLogic?
          (bQuantifier '.')?
          BID
          ( ( '.' BID ) | ( '.' (bQuantifier '.')? ( bWrapSpaceType
| bWrapStructureType ) ) | ( '.' BFunction) )+
          bComparisonOperator?
          (  BID  |  bStringQuot  |  '-'?  INTLITERAL  |  '-'?
DOUBLELITERAL )?
    ;

// TODO
bBOMExpr
    :
        ((BQuantifier '.')? BID '.')?

```



```

                ((BQuantifier '.')? (bWrapSpaceType '.' |
bWrapStructureType '.'))+
                (BQuantifier '.')? (BID | BFunction)+
                ('.' BFunction)?
            ;

bQuantifier
    :    BQuantifier
    ;

bComparisonOperator
    :    ( '=' | '==' | '>' | '<' | '>=' | '<=' )
        |    bComparisonOperatorNegation
    ;

bComparisonOperatorNegation
    :    ( '!=' | '! =' | '!==' | '! ==' )
    ;


// *****
// Rule Definition
bRuleDef
    :    (bRuleDefUnit LT*)+
    ;

bRuleDefUnit
    :    Bdefine? bDefType (':' bRuleType)? BID '(' bParamDef?
        ')' (EXTENDS BID)?
        '{' LT* bRuleDefExpr* '}'
    ;

```

```

bRuleDefExpr
    :      bBOMPropExpr | bBOMDecLine
    ;

bParamDef
    :      (bWrapSpaceType      |      bWrapStructureType)      BID      (','
(bWrapSpaceType | bWrapStructureType) BID)*
    ;


// *****

// BERA Execution

bExeStat
    :      (bExeStatUnit)*
    ;

bExeStatUnit
    :      bRuleExeLines
    |      bExeIfThenElseStat
    ;

bRuleExeLines
    :      bRuleExeLine ';'
    ;

bRuleExeLine
    :      (      bRuleVerb      |      BID      )      '('      aa=bBlockExpr      (','
bBlockExpr)?      ')'
        // e.g. myRule("office");
        //          e.g.          myRule(getSpace("toilet"));
        myRule(getSpace("toilet") + getSpace("MEN"));
    ;

```

```

bBlockExpr
    :      //bBlockPEExprUnit (',' bBlockPEExprUnit )*
        BID
    |      bWrapSpaceType
    |      bWrapStructureType
    |      bStringQuot
    |      bBOMGetter
    ;

```

```

bRuleVerb
    :      Bget | Bcheck
    ;

```

```

bExeIfThenElseStat
    :      bExeIfStat bExeThenStat bExeElseStat? ';'
    ;

```

```

bExeIfStat
    :      IF '(' bRuleExeLine ')' LT*
    ;

```

```

bExeThenStat
    :      bRuleExeLine LT*
    ;

```

```

bExeElseStat
    :      ELSE bRuleExeLine
    ;

```

```

// *****

```

```

// Wrapper Data Type in BERA

```

```

bWrapSpaceType
    :      BBuilding
    |      BFloor
    |      BSpace
    |      BSpaceGroup
    |      BPath
    ;

// Extensible Structure Keywords
bWrapStructureType
    :      BStructure
    |      BSlab
    |      BColumn
    |      BWall
    |      BWindow
    |      BDoor
    |      BStair
    |      BRamp
    |      BCurtainWall
    |      BRoof
    ;

// *****
// Common & Link to Lexer
bStringRep
    :      BID (',' BID)*
    ;

bStringQuotRep
    :      bStringQuot (',' bStringQuot)*
    ;

```

```

bStringQuot
    :      '""' ( BID | INTLITERAL )+ '""'
    ;

```

```

bVerbs
    :      Bdefine
    |      Bget
    |      Bcheck
    ;

```

```

bDefType
    :      BRule
    ;

```

```

bRuleType
    :      BArea
    |      BProgram
    |      BCirculation
    |      BSecurity
    ;

```

```

bLogic
    :      AND | OR
    ;

```

```

/*****

    BERA LEXER DEFINITION AREA

*****/

// *****/

// BERA Keywords

BERABEGIN :      'BERABEGIN' ;

```

```

BERAEND      :      'BERAEND' ;

Breference   :      'reference' | 'REFERENCE'
              ;

BbuildingType :      'buildingType'      |      'buildingtype'      |
'BUILDINGTYPE'
              ;

BQuantifier  :      'all' | 'ALL'
              |      'one' | 'ONE'
              |      'two' | 'TWO'
              ;

BFunction    :      'count' | 'min' | 'max'
              |      'average' | 'median' | 'round'
              ;

BProtocol    :      ('http://' | 'HTTP://')
              |      ('https://' | 'HTTPS://' )
              |      ('ftp://' | 'FTP://' )
              ;

// BOM - Spatial Keywords

BBuilding    :      'Building' ;

BFloor       :      'Floor' ;

Bspace       :      'Space' ;

BspaceGroup  :      'SpaceGroup' ;

BPath        :      'Path' ;

// BOM - Extensible Structure Keywords

BStructure   :      'Structure' ;

```

```

BDoor      :      'Door' ;
BSlab      :      'Slab' ;
BColumn    :      'Column' ;
BWall      :      'Wall' ;
BCurtainWall :      'CurtainWall' ;
BStair     :      'Stair' ;
BRamp      :      'Ramp' ;
BWindow    :      'Window' ;
BRoof      :      'Roof' ;

// BERA Rule & Rule types
BRule      :      'Rule' ;
BArea      :      'Area' ;
BProgram   :      'Program' ;
BCirculation :      'Circulation' ;
BSecurity  :      'Security' ;

// BERA Execution Verbal Keywords
Bget       :      'get' ;
Bdefine    :      'define' ;
Bcheck     :      'check' ;

// Shortcut keywords - BERA Execution
BgetBuilding :      'getBuilding' ;
BgetSpace   :      'getSpace' ;
BgetFloor   :      'getFloor' ;
BgetSpaceGroup :      'getSpaceGroup' ;
BgetPath    :      'getPath' ;

BdefineRule :      'defineRule' ;

```

```

BdefineSpace      :      'defineSpace' ;
BdefineFloor      :      'defineFloor' ;
BdefineSpaceGroup
                  :      'defineSpaceGroup' ;
BdefineZone       :      'defineZone' ;
BdefinePath       :      'definePath' ;


BcheckRule :      'checkRule' ;
BcheckBuilding :      'checkBuilding' ;
BcheckFloor :      'checkFloor' ;
BcheckSpace :      'checkSpace' ;
BcheckSpaceGroup
              :      'checkSpaceGroup' ;
BcheckZone :      'checkZone' ;
BcheckPath :      'checkPath' ;


BgetStructure :      'getStructure' ;
BgetDoor :      'getDoor' ;
BgetSlab :      'getSlab' ;
BgetColumn :      'getColumn' ;
BgetWall :      'getWall' ;
BgetStair :      'getStair' ;
BgetRamp :      'getRamp' ;


// System Execution keywords
Bprint :      'print' ;
Bprintln :      'println' ;
Bvisualize :      'visualize' ;
//Breport :      'report' ;

```



```

// Java keywords

EXTENDS      :      'extends' | 'EXTENDS' ;

IF           :      'if' | 'IF' ;

ELSE        :      'else' | 'ELSE' ;

FOR         :      'for' | 'FOR' ;

OR          :      'or' | 'OR' ;

AND         :      'and' | 'AND' ;


// *****

// Identifiers for variable names..

BID         :      BIDprefix ( BIDprefix | BIDDigit )* ;
fragment
BIDprefix   :      'a'..'z'|'A'..'Z'|'_' ;
fragment
BIDDigit    :      '0'..'9' ;


// *****

// System Tokens

INTLITERAL :

            IntegerNumber

            ;

fragment
IntegerNumber
            : '0'
            | '1'..'9' ('0'..'9')*
            | '0' ('0'..'7')+
            ;

```

DOUBLELITERAL

```
:   NonIntegerNumber DoubleSuffix?
;
```

fragment

NonIntegerNumber

```
:   ('0' .. '9')+ '.' ('0' .. '9')* Exponent?
|   '.' ('0' .. '9')+ Exponent?
|   ('0' .. '9')+ Exponent
|   ('0' .. '9')+
|
    HexPrefix (HexDigit )*
    (
        (
            |   ('.' (HexDigit )* )
        )
        ( 'p' | 'P' )
        ( '+' | '-' )?
        ( '0' .. '9' )+
    );
```

fragment

DoubleSuffix

```
:   'd' | 'D'
;
```

fragment

Exponent

```
:   ( 'e' | 'E' ) ( '+' | '-' )? ( '0' .. '9' )+
;
```

fragment

HexDigit

```
:   ('0'..'9'|'a'..'f'|'A'..'F')
```

```
    ;  
fragment  
HexPrefix  
    :    '0x' | '0X'  
    ;
```

APPENDIX D

BERA LANGUAGE USER MANUAL

The BERA Language User Manual is given to users through the BERA website for its up-to-date version. In terms of its technical development, development work has just begun: it is open-ended and still growing. The initial release of the BERA Language and BERA Language Tool is version 1.0 as of 2010 fall. The BERA Language User Manual focuses on describing BERA language semantics: what BERA Language and its Tool are and how to write BERA language programs, rather than explaining detailed context-free grammar-based language syntax and its low-level execution procedures. We expect the on-line User Manual to help the users to grasp the overall features of BERA Language, and actually write/test their programs.

The website URL is:

<http://bim.arch.gatech.edu/bera>

Contacts for developers:

Jin Kook Lee, leejinkook@gmail.com

Chuck Eastman, Professor, chuck.eastman@coa.gatech.edu

REFERENCES

- ActionScript. “Adobe Inc., Action Script”, a scripting language based on ECMA script for controlling Adobe Flash Media (FLA/SWF files). Available: <http://www.actionscript.org/> (Accessed Feb 2010)
- Adachi, Y. “Overview of partial model query language”, in: Proc. of the 10th Int. Conf. on Concurrent Engineering, 2003.
- Adachi, Y. “Research on Building Information Model and IFC”, Yoshinobu Adachi, Building Technology Group, Available: <http://i-marina.secom.co.jp/isl/e2/research/cs/report02/index.html> (Accessed Feb 2010)
- ANSI/BOMA. “Standard Method for Measuring Floor Areas in Office Buildings”, Building Owners and Managers Association International, ANSI/BOMA Z65.1-1996. 1996.
- AIA. “Guidelines for Design and Construction of Health Care Facilities”, The American Institute of Architects Press, 2006.
- AIA. “Guidelines for design and construction of Hospital and Health care Facilities”, The American Institute of Architects Press, Washington D.C., 1997.
- AISC. “CIS/2: CIMSteel Integration Standards Release 2”, Available: <http://www.cis2.org>. (Accessed Nov 2010)
- Arduino. “Arduino Language”, an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. Available: <http://www.arduino.cc> (Accessed Feb 2010)
- ArrayList. “A Java collection class ArrayList”, Available: <http://download.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html>. (Accessed Nov 2010)
- Autodesk. “Autodesk Revit Architecture”, one of the major BIM authoring tools. Available: <http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=3781831> (Accessed Oct 2010)

- Bentley Systems. “Bentley”, one of the major BIM authoring tools. Available: <http://www.bentley.com/en-US/Products/Bentley+Architecture/> (Accessed Oct 2010)
- Booch, G., Rumbaugh J, Jacobson I. “The Unified Modeling Language User Guide”, Addison-Wesley Professional, ISBN 978-0201571684, 1998.
- Borrmann, A., Treeck C V, and Rank E. “Towards a 3D spatial query language for building information models”, in: Proc. of the Joint Int. Conf. for Computing and Decision Making in Civil and Building Engineering, 2006.
- Borrmann, A., Schraufstetter S, Rank E. “Implementing Metric Operators of a Spatial Query Language for 3D Building Models: Octree and B-Rep Approaches”, Journal of Computing in Civil Engineering 23 (1), pp. 34-46. 2009a.
- Borrmann, A., Rank E. “Specification and implementation of directional operators in a 3D spatial query language for building information models”. Advanced Engineering Informatics 23 (1), pp. 32-44. 2009b.
- Borrmann, A. “3D Spatial Query Language for Building Information Models”, N. Paul, A. Borrmann, Available: <http://www.cie.bv.tum.de/index.php/de/component/content/article/58>. (Accessed Nov 2010a)
- Borrmann, A. “Intelligent construction rule checking using a Spatial Constraint Language”, A. Borrmann (TUM), J. Hyvärinen, T. Mäkeläinen (VTT), Available: <http://www.cie.bv.tum.de/index.php/de/component/content/article/61>. (Accessed Nov 2010b)
- buildingSMART. “Industry Foundation Classes (IFC)”. Available: www.buildingsmart.com/bim (Accessed Feb 2010a)
- buildingSMART. “The EXPRESS Definition Language for IFC Development”, Available: <http://www.civ.utoronto.ca/sect/coneng/i2c/Civ1283/Civ1283-Ref-Final/Civ1283-Basic%20Ref/IFC/Express-101.pdf> (Accessed Oct 2010b)
- Chamberlin, Donald D., Boyce, Raymond F. “SEQUEL: A Structured English Query Language” Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control (Association for Computing Machinery): 249–

264. Available: <http://www.almaden.ibm.com/cs/people/chamberlin/sequel-1974.pdf>. (Accessed Feb 2010), 1974.

Dijkstra, Edsger W. "On the role of scientific thought". in Dijkstra, Edsger W. Selected writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag New York, Inc. pp. 60–66. ISBN 0-387-90652-5. 1982.

Dijkstra, Edsger W. "Dijkstra's Algorithm", a shortest path finding algorithm invented by Dijkstra in 1956. Available: http://www.algolist.com/Dijkstra's_algorithm. (Accessed Nov 2010)

Duckham, M., L.K. "Simplest Paths: Automated Route Selection for Navigation", Lecture Notes in Computer Science Vol. 2825: Spatial Information Theory, 169–185, 2003.

Dym, C. L., Henchey R. P., Delis E. A., and Gonick S. "A knowledge-based system for automated architectural code checking", Computer-Aided Design Volume 20, Issue 3: 137-145, 1998.

Eastman, C. M. "Building Description System BDS-10 User's Manual", Institute of Physical Planning, Carnegie Mellon University, 1976.

Eastman, C. M. "Building Product Models: Computer Environments Supporting Design and Construction", CRC Press, 1999.

Eastman, C. M. "Enumerating architectural arrangements by generating their underlying graphs", Environment and Planning B: Planning and Design vol. 7, 289-310, 1980.

Eastman, C. M. "From blue print to database", Economist, Economist Technology Quarterly, June 7, pp18-22, 2008.

Eastman, C. M. "General Purpose Building Description Systems", Computer-Aided Design, Volume 8, Issue 1, Pages 17-26, 1975.

Eastman, C. M. "Heuristic Algorithms for automated space planning". IJCAI 1971: 27-39. 1971.

- Eastman, C. M. "IFC Overview", Chuck Eastman, 2006. Available: <http://bim.arch.gatech.edu> (Accessed Feb 2010)
- Eastman, C. M., Lee J, Jeong Y-s, and Lee J-K. "Automatic Rule-Based Checking of Building Designs", *Automation In Construction*, Vol.18 Issue 8, 1011-1033, 2009b.
- Eastman, C. M., Lee J-K, Sheward H, Sanguinetti P, Jeong Y-s, Lee J, and Abdelmohsen S. "Automated Assessment of Early Concept Designs", *AD (Architectural Design)* Vol 79. No 2. "Closing the Gap – Information Models in Contemporary Design Practice." John Wiley & Sons Inc. 2009a.
- Eastman, C. M., and Henrion M. "Glide: A Language for Design Information Systems", *ACM SIGGRAPH Computer Graphics archive*, Volume 11 , Issue 2, 1977.
- Eastman, C. M., Teicholz P, Sacks R, and Liston K. "BIM Handbook – A guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors", John Wiley & Sons Inc. 2008.
- ECMA International. "ECMA International", Available: <http://www.ecma-international.org/> (Accessed Feb 2010a)
- ECMA International. "Standard ECMA-262 - ECMAScript Language Specification", 2009, Available: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf> (Accessed Feb 2010b)
- Egenhofer, M. J. "An extended SQL syntax to treat spatial objects", in: *Proc. of the 2nd Int. Seminar on Trends and Concerns of Spatial Sciences*, 1987.
- Egenhofer, M. J. "Spatial SQL: A Query and Presentation Language", *IEEE Transactions on Knowledge and Data Engineering* 6 (1): 86-95, 1994.
- Elmasri, R., and Navathe S B. "Fundamentals of Database Systems." New York, Addison Wesley, 2003.
- Eurostep. "Eurostep Share-A-space Server Solution", Available: <http://www.eurostep.com/global/solutions/--share-a-space-server-solution.aspx> (Accessed Oct 2010)

Fritzing. “Fritzing Language”, an open-source initiative to support designers, artists, researchers and hobbyists to take the step from physical prototyping to actual product. Available: <http://fritzing.org> (Accessed Feb 2010)

Gamma, E., Helm R, Johnson R, and Vlissides J. “Design Patterns”, Addison-Wesley Inc. 1995.

Gay, J. “The History of Flash”, Macromedia Showcase, Jonathan Gay, 2001, Available: http://www.adobe.com/macromedia/events/john_gay/page02.html (Accessed Feb 2010)

Goodrich, M. T., Tamassia R. “Data Structures and Algorithms in Java”, ISBN: 0-471-73884-0, John Wiley & Sons, Inc., 2005.

Graphisoft. “Graphisoft ArchiCAD”, one of the major BIM authoring tools. Available: <http://www.graphisoft.com/products/archicad/> (Accessed Oct 2010)

Gross, J., and Yellen J, “Graph Theory and its Applications”, Discrete Mathematics Series, CRC Press, 1998.

GSA-GT. “GSA BIM-enabled Design Guide Automation”, Available: <http://dcom.arch.gatech.edu/gas>. (Accessed Feb 2010)

GSA. “GSA BIM Guide Series 2: Spatial Program Validation”, U.S. GSA, Available: www.gsa.gov/bim (Accessed Feb 2010a)

GSA. “National Business Space Assignment Policy”, U.S. GSA Public Building Service, Available: <http://www.gsa.gov/portal/content/102002>. (Accessed Feb 2010b)

Herring, J., Larsen R, and Shivakumar J. “Extensions to the SQL language to support spatial analysis in a topological data base”, in: Proc. of GIS/LIS’88, 1988.

HOPL. “HOPL: the History of Programming Languages”, Available: <http://hopl.murdoch.edu.au/> (Accessed Feb 2010)

Hwang, E. “Digital Logic and Microprocessor Design with VHDL”, Thomson. ISBN 0-534-46593-5. Available: <http://faculty.lasierra.edu/~ehwang/digitaldesign> (Accessed Nov 2010)

- IAI. “Industry Foundation Classes Release 2x”, IFC Technical Guide: International Alliance for Interoperability. 2000.
- IAI. . “Industry Foundation Classes Release 2x Edition 2”, IFC Technical Guide: International Alliance for Interoperability. 2003.
- ICC. “ICC: International Code Council), Available: <http://www.iccsafe.org> (Accessed Sep 2010)
- ICC. “SmartCodes Examples”, ICC, Available: <http://www2.iccsafe.org/io/smartcodes> (Accessed Sep 2010)
- IFC 2x3TC1. “IFC 2x Edition 3 Technical Corrigendum 1 Specification”, buildingSMART, Available: <http://www.iai-tech.org/ifc/IFC2x3/TC1/html/index.htm> (Accessed Sep 2010)
- Ingram, K., and Phillips W. “Geographic information processing using a SQL-based query language”, in: Proc. of the 8th Int. Symp. on Computer-Assisted Cartography, 1987.
- ISO 10303. “ISO 10303-11:1994”, Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual, Available: http://www.iso.org/iso/catalogue_detail?csnumber=18348 (Accessed Feb 2010)
- ISO 14977. “ISO/IEC 14977:1996”, Information technology - Syntactic metalanguage - Extended BNF, Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153 (Accessed Feb 2010)
- ISO 16262. “ISO/IEC 16262:2002, Information technology - ECMAScript language specification”, Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=33835 (Accessed Feb 2010)
- Java. “Java Language and its tools and resources”, Oracle Inc, Available: <http://www.java.com/en/> (Accessed Nov 2010)

- Johnson, S. C. "Yacc: Yet another compiler compiler." In UNIX Programmer's Manual, volume 2, pp 353-387. Holt, Rinehart, and Winston. New York, NY, US, 1979.
- jQuery. "jQuery Project", a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. Available: <http://jquery.org/> (Accessed Nov 2010)
- Kannala, M. "Escape Route Analysis Based on Building Information Models: Design and Implementation". Department of Computer Science and Engineering, Helsinki University of Technology. M.S. Thesis, 2005.
- Kent, Stuart. "Model Driven Engineering", Integrated Formal Methods, Lecture Notes in Computer Science, Volume 2335/2002, 286-298, DOI: 10.1007/3-540-47884-1_16, 2002.
- Khan, Sohail Zaffar, "Comparison of Selection Criteria of Open Source DBMS Against Proprietary DBMS", Master's Thesis in Advanced Financial Information Systems, Swedish School of Economics and Business Administration, 2006.
- Lee, G. "A new formal and analytical process to product modeling (PPM) method and its applications to the Precast Concrete Industry", PhD dissertation of College of Architecture, Georgia Institute of Technology, 2004.
- Lee, J, Eastman C M, Jeong Y-s, Lee J-K, and Sheward H, "Automated Rule Checking of Security and Circulation Using IFC Building Model", Under review, 2010.
- Lee, J-K, Eastman C M, Lee J, and Kannala M, Jeong Y-s, "Computing Walking Distances within Buildings based on the Universal Circulation Network", Environment and Planning B (EPB): Planning and Design, 37(4) 628 - 645, 2010a.
- Lee, J-K, Lee J, Jeong Y-s, Sheward H, Sanguinetti P, Abdelmohsen S, and Eastman C M, "Development of Space Object Semantics for Automated Building Design Review Systems", Under review, 2010b.
- Lesk, M.E., "Lex – a lexical analyzer generator." Technical Report Computing Science Technical Report No.39, Bell Telephone Laboratories, 1975.
- Lethbridge, T. C., and Laganieri R. "Object-oriented Software Engineering", McGraw-Hill, ISBN 978-0073220345, 2005.

- Levenshtein. "Levenshtein Algorithm", Available: <http://www.levenshtein.net>. (Accessed Feb 2010)
- Liu, H. "Unpacking Meaning from Words: A Context-Centered Approach to Computational Lexicon Design", Lecture Notes in Computer Science, Volume 2680/2003, pp 218-232, 2003.
- Mashey, J. R. "New programming languages are born every day. Why do some succeed and some fail?", ACM Queue Volume 2, Issue 9 (December/January 2004-2005) - Languages, Levels, Libraries, and Longevity, 2004.
- MDA. "Model driven architecture (MDA)", OMG Architecture Board ORMSC, OMG document number ormsc/2001-07-01, 2001.
- Metsker. "Building Parsers with Java", ISBN-13: 978-0201719628, Addison-Wesley Professional, 2001.
- MIT Media Lab. "Design by Numbers (DBN) Language", a language created for visual designers and artists as an introduction to computational design. Available: <http://dbn.media.mit.edu/> (Accessed Feb 2010)
- MVD. "The Model View Definition", IFC Solutions Factory, Available: <http://www.blis-project.org/IAI-MVD>. (Accessed Feb 2010)
- Netscape Communications. "The Netscape Archive", Available: <http://browser.netscape.com/> (Accessed Feb 2010)
- NFPA. "NFPA 101: Life Safety Code", Measurement of Travel Distance to Exits, NFPA101.7.6 and chapters 12-42, 2006.
- NY. "Building Code of the City of New York", Department of Citywide Administrative Services NY, 2004.
- OCCS. "OmniClass Construction Classification System", OCCS Table 13: Spaces by Function, and 14: Spaces by Form, CSI: The Construction Specifications Institute, 2010.

- Ooi, R. Sacks-Davis, and McDonell K, “Extending a DBMS for geographic applications”, in: Proc. of the IEEE 5th Int. Conf. on Data Engineering, 1989.
- OSHA. “Occupational Safety and Health Administration”, Maintenance, Safeguards, and Operational Features for Exit Routes, Available: <http://www.osha.gov/SLTC/etools/evacuation/egress.html> (Accessed Oct 2010)
- Parr, T, “ANTLR”, Another tool for language recognition, Available: [www.antlr.org](http://wwwantlr.org) (Accessed Feb 2010a)
- Parr, T, “StringTemplate”, Available : www.stringtemplate.org (Accessed Feb 2010b)
- Parr, T. “Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages”, Pragmatic Bookshelf, ISBN: 978-1934356456, 2009.
- Parr, T. “Soapbox: Humans should not have to grok XML - Answers to the question When shouldn't you use XML?”, IBM developerWorks, Terence Parr, 2001, Available: <http://www.ibm.com/developerworks/xml/library/x-sbx.xml.html> (Accessed Feb 2010c)
- Parr, T. “The Definitive ANTLR Reference: Building Domain-Specific Languages”, Pragmatic Bookshelf, ISBN: 978-0978739256, 2008.
- Peponis, J., Wineman J, Bafna S, Rashid M, Kim S H. “On the generation of linear representations of spatial configuration”, Environment and Planning B: Planning and Design 25, 559 – 576, 1998.
- Processing. “Processing Language”, an open source programming language and environment for people who want to program images, animation, and interactions. Available: <http://www.processing.org/> (Accessed Feb 2010)
- Renz, J. “Qualitative Spatial Reasoning with Topological Information”, Springer, Verlag, 2002.
- Roussopoulos, N., Faloutsos C, and Sellis T, An efficient pictorial database system for PSQL, IEEE Transactions on Software Engineering 14 (5) 639–650, 1988.

Sanguinetti, P., Abdelmohsen S, Lee J, Lee J-K, Sheward H, and Eastman C M, “General System Architecture Approach to BIM Models”, Under submission, 2010.

Schultz, C., Amor R, Lobb B, and Guesgen H, “Qualitative design support for engineering and architecture”, *Advanced Engineering Informatics* 23 (1) 68–80, 2008.

Scott, M.L. “Programming Language Pragmatics, Second Edition”. Morgan Kaufmann Publishers, ISBN 978-0126339512, 2005.

Solibri, “Solibri Model Checker® (SMC)”, an IFC-based rule checking BIM tool.
Available: <http://www.solibri.com> (Accessed Feb 2010)

SQL. “Structured Query Language (SQL)”. International Business Machines. October 27, 2006. Available:
<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004100.htm>. (Accessed Feb 2010)

The Open Group. “Regular Expressions”, The Single UNIX Specification, Version 2, 1997, Available: <http://www.opengroup.org/onlinepubs/007908799/xbd/re.html> (Accessed Feb 2010)

TIOBE Soft. “TIOBE Programming Community Index”, TIOBE software, Available:
<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html> (Accessed Feb 2010)

US Access Board. “ADAAG, Americans with disabilities act accessibility guide”, 2002.

US Access Board. “Justice for All: Designing Accessible Courthouses”, 2006.

US Access Board. “ADA Accessibility Guidelines (ADAAG)”, 2004, A Federal Agency Committed to Accessible Design. Available: <http://www.access-board.gov> (Accessed Feb 2010)

USCDG. “U.S. Courts Design Guide 1997”, Judicial Conference of the United States, Space and Facilities Division. 1997.

USCDG. “U.S. Courts Design Guide 2007”, Judicial Conference of the United States, Space and Facilities Division. 2007.

W3C. “World Wide Web Consortium: W3C”, An international community where member organizations, a full-time staff, and the public work together to develop Web standards. Led by Web inventor Tim Berners-Lee and CEO Jeffrey Jaffe. Available: <http://www.w3.org/> (Accessed Nov 2010a)

W3C. “W3C Cascading Style Sheets (CSS)”, Available: <http://www.w3.org/TR/CSS1/> (Accessed Feb 2010b)

W3C. “W3C Document Object Model (DOM)”, Available: <http://www.w3.org/DOM/> (Accessed Feb 2010c)

W3C. “Extensible Hyper Text Markup Language 2 (XHTML2) Working Group Home Page”, also contains old standards for HTML4, XHTML1.0, etc. Available: <http://www.w3.org/MarkUp/> (Accessed Nov 2010d)

W3C. “Extensible Markup Language: XML”, Available: <http://www.w3.org/XML/> (Accessed Nov 2010e)

W3C. “Scripting and AJAX”, Available: <http://www.w3.org/standards/webdesign/script.html> (Accessed Nov 2010f)

WBDG. “WBDG: Whole Building Design Guide”, Available: <http://www.wbdg.org> (Accessed Sep 2010)

Web2.0. “Web 2.0 by Paul Graham: Does "Web 2.0" mean anything?”, Available: <http://www.paulgraham.com/web20.html> (Accessed Nov 2010)

Werner, S., B.K.-B., Herrmann T. “Modelling Navigational Knowledge by Route Graphs”, Spatial Cognition II, LNAI 1849, 295-316, 2000.

Wikimedia. “Wikipedia”, a free web-based encyclopedia. Available: <http://wikimediafoundation.org/wiki/Home> (Accessed Feb 2010)

Wiring. “Wiring Language”, an open source programming environment and electronics i/o board for exploring the electronic arts, tangible media, teaching and learning

computer programming and prototyping with electronics. Available:
<http://wiring.org.co> (Accessed Feb 2010)

XHTML. "Extensible Hyper Text Markup Language (XHTML) Reference", Available:
<http://xhtml.com/en/xhtml/reference/> (Accessed Nov 2010)

Yang, D., Yoo S, Wang F, Eastman C M, Lee J. "CIS/2 at Georgia Tech", Available:
<http://dcom.arch.gatech.edu/old/cis2ifc/>. (accessed Nov 2010)

Zhi, G.S., S. M. L., Z. Fang. "A graph-based algorithm for extracting units and loops from architectural floor plans for a building evacuation model." *Computer-Aided Design* 35, 2003.

VITA

LEE, JIN KOOK

Mr. Jin kook Lee received a bachelor's degree in the department of Housing and Interior Design from Yonsei University, Seoul, Korea in 2000. Upon leaving under-grad school, he was employed at a major housing design and manufacturing company Hanssem Co. Ltd, and he was one of the co-founders (one of five) and a developer of the IT Company in so-called venture-valley in Seoul. He had worked for several IT-based projects within the research domain of "design & computation". He was back to Yonsei University and studied at the Digital Design Media Lab and Institute of Media Art. He had published seven (inter)national conference papers, joined several government and industry projects in three research institutes, and received seven design computation related awards and scholarships during and after his M.S. study.

He began his Ph.D. study at Georgia Tech College of Architecture in Aug. 2005, and joined the Design Computing program and Digital Building Lab (formerly AEC Integration Lab) led by his advisor Professor Charles M. Eastman. Not only as one of the members for the US Federal government GSA funded project that is still ongoing since 2006, but also as a Design Computing-majored student, he has researched and developed several software on the theoretical and technical basis of BIM (Building Information Modeling) thanks to his advisor and research colleagues. Automated space program review for the US Courthouses, spatial database system to support space object semantics, metric graph algorithm for the building circulation, GT energy performance standard calculation toolkit, BIM Resources website, several project websites, and the BERA Language design and its implementation are the software products developed by him.

During his PhD studies, he published four journal and two conference papers as the first and one of the co-authors. Several non-refereed project reports and articles have been published and some journal papers are under review. He and his project team received "BIM Award Citation at 2008 AIA TAP BIM Award" from the AIA (American Institute of Architects) based on their research and development project funded by the US GSA.