MULTI-LAYER CELLULAR DEVS FORMALISM FOR FASTER MODEL DEVELOPMENT AND SIMULATION EFFICIENCY

by

Fahad Awadh Saleem Bait Shiginah

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2006

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation

prepared by Fahad A. Bait Shiginah

entitled Multi-layer Cellular DEVS Formalism for Faster Model Development and Simulation Efficiency

and recommend that it be accepted as fulfilling the dissertation requirement for the

Degree of Doctor of Philosophy

____ Date: November 17, 2006
Bernard P. Zeigler, Ph.D.

____ Date: November 17, 2006
Jerzy W. Rozenblit, Ph.D.

____ Date: November 17, 2006
Salim A. Hariri, Ph.D.

____ Date: November 17, 2006
Moon-Ho Hwang, Ph.D.

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

____ Date: November 17, 2006
Dissertation Director: Bernard P. Zeigler, Ph.D.

STATEMENT BY AUTHOR

ACKNOWLEDGEMENTS

First of all, I thank my advisor Dr. Bernard P. Zeigler for providing me with all support and guidance in my Ph.D. study. A letter of appreciation goes to this great professor whom I will continue learning from, for the rest of my life. Besides my advisor, I would like to thank the rest of my dissertation committee: Dr. Salim Hariri, Dr. Jerzy Rozenblit and Dr. Moon-Ho Hwang for their helpful comments, suggestions and encouragement during the course of completing this dissertation. Special thanks go to Dr. Moon-Ho Hwang, for his help, discussions, and encouragement to my research ideas.

Most of all, my wife (accompanying me), parents, brothers, and sisters (in Oman) are the ones who have been with me and I always remember their constant love and great support. I am grateful to my wife, Muna, for her patience, support, and love especially in taking care of me as well as our children during pursuing this research.

I thank all my colleagues in the ACIMS Lab for the such wonderful environment. Special thanks go to Raj, Brett, and Eddie Mak.

Finally, again, thank you Muna.

DEDICATION

To

my parents,

my wife Muna,

my kids, and all family members

TABLE OF CONTENTS

LIST OF FIGURES 10

LIST OF TABLES 12

ABSTRACT 13

CHAPTER 1 : INTRODUCTION 15

1.1 Motivation and General Scope 15

1.2 Main Contributions 17

1.3 Dissertation Outline 18

CHAPTER 2 : BACKGROUND 20

2.1 Cellular Automata 20

2.2 Parallel DEVS Formalism (P-DEVS) 21

2.2.1 Closure under Coupling of Parallel DEVS 23

2.3 Multi-Component DEVS Formalism 25

2.4 Cellular Space Models in DEVS 27

2.4.1 Closure under Coupling for Cellular Models in Parallel DEVS 29

2.5 Related Work 30

2.5.1 Cell-DEVS Formalism 30

2.5.2 Converting Coupled Model into Atomic DEVS 32

CHAPTER 3 : NEW FRAMEWORK FOR CELLULAR DEVS MODELING 34

3.1 Converting Cell Space Model into Atomic DEVS 34

TABLE OF CONTENTS - *Continued*

3.2 Toward Full Decomposition of Cell Space Models.. 35

3.3 Closure under Coupling of Parallel DEVS Applied to Cell Spaces 35

3.3.1 Event List Handling 37

3.3.2 Transforming Cells to Non-Modular Form.. 42

3.3.3 Final Non-Modular Decomposed Format 51

3.4 A Proposition to Show the Generality of the Approach 56

3.5 Fast Cellular DEVS Specification 58

3.6 Solving Differential Equations Using Cell Space Models............................... 66

3.7 Solving Partial Differential Equations Using the New Framework................. 69

CHAPTER 4 : A TOOL FOR BUILDING EFFICIENT CELLULAR DEVS MODELS 72

4.1 Ease of User Specifications with GUI support 72

4.2 Design Structure of the New Modeling Environment 75

4.3 New Cellular DEVS Specification Unit 77

4.3.1 DEVS Cell Space Implementation Models.. 78

4.3.2 Some Implementation Details in Cellular DEVS Models 79

4.4 The GUI 84

4.5 The Code Generator.. 86

4.5.1 The Generated Model Classes 87

4.5.2 Some Implementation Issues in the Code Generator................................ 88

TABLE OF CONTENTS - *Continued*

CHAPTER 5 : DESIGN ISSUES OF THE EVENT LIST 91

5.1 Event List Design Requirements 92

5.2 Standard Array Implementation 97

5.3 Binary Heap Implementation 99

5.4 Binary Tree Implementation 101

5.5 Analytical Comparison 103

5.6 Experimental Analysis 108

5.7 Concluding Remarks 112

CHAPTER 6 : TESTING AND VERIFICATION 114

6.1 A Quick Overview on Software Testing Techniques 114

6.2 A Quick Overview on Simulation Model Validation and Verification 115

6.3 Validation and Verification of DEVS Models 116

6.4 Testing the Cellular DEVS Specification Unit 117

6.4.1 Testing *cell* and *block* Classes 118

6.4.2 Testing *cellSpace* and *blockSpace* Classes 120

6.5 Testing the Generated Models 120

6.5.1 Generated Code Test 121

6.5.2 Atomic Cell DEVS Verification 122

6.6 Verifying the Approach 126

6.7 Applying All Tests 127

TABLE OF CONTENTS - *Continued*

CHAPTER 7 : LANDSLIDE APPLICATION MODELS 131

7.1 Overview on Landslides and the Need for Their Models 131

7.2 Cellular DEVS Models for Landslides 132

7.3 Non-Timed Cellular Automata Landslide Model 133

7.4 Discrete Time Cellular Landslide Model... 134

7.5 Discrete Event Cellular Landslide Model.. 136

7.6 Rate-Based Predictive Quantized Landslide Model 137

7.7 Models Development Experiences 139

7.8 Experimental Results 141

7.8.1 Modular vs. Non-Modular Approach 141

7.8.2 Using the New Implementation for Conventional Cellular Models 143

7.8.3 Different Blocking Setting (Living with Messages)............................... 144

CHAPTER 8 : CONCLUSIONS AND FUTURE WORK 146

REFERENCES 151

LIST OF FIGURES

Figure 2.1: 2-D cellular space in DEVS. 28

Figure 3.1: Switching between modular and non-modular forms. 43

Figure 4.1: Specification layers. 75

Figure 4.2: Structure of the new cellular DEVS environment. 76

Figure 4.3: Cellular DEVS specification unit. 78

Figure 4.4: Supported neighboring rules. 79

Figure 4.5: Neighbors addressing convention. 80

Figure 4.6: Single port labeling scheme. 82

Figure 4.7: Multi port labeling scheme. 83

Figure 4.8: The main GUI view. 85

Figure 4.9: The cell transition function tab. 85

Figure 4.10: Structure of the code generator unit. 86

Figure 4.11: Generated model classes and hierarchy. 87

Figure 4.12: Illustration of code generation process. 89

Figure 5.1: Event list in unsorted array. 97

Figure 5.2: Event list in sorted array. ($ta_x > ta_y$ for any $x<y$) 98

Figure 5.3: Binary heap structure. ($ta_x \le ta_y$ for any child y with parent x) 100

Figure 5.4: Binary heap as an array. 100

Figure 5.5: Balanced binary search tree. ($ta_4 < ta_2 < ta_1 < ta_3 < ta_n$) 102

Figure 6.1: Simplified modeling process. Taken from [61] 116

LIST OF FIGURES - *Continued*

Figure 6.2: Examples of using Java reflection in testing unit class. 122

Figure 6.3: Cell verification experimental frame... 123

Figure 6.4: Cell verification experimental frame with special generator. 125

Figure 6.5: Simulation approach of equivalency verification.. 126

Figure 6.6: Beginning of specification unit test result.. 127

Figure 6.7: End of specification unit test. 128

Figure 6.8: An example of cell verification test report. 130

Figure 6.9: An example of non-identical spaces... 130

Figure 6.10: An example of identical spaces. 130

Figure 7.1: A representation of hexagonal land cell that accounts for energy using the kinetic head. 133

Figure 7.2: Directional vectors of cell neighbors ($\vec{J_v}$) in hexagonal lattice. 136

Figure 7.3: Slope vector calculation. 136

Figure 7.4: An example of incorrectness and instability. 138

LIST OF TABLES

Table 4.1: Single vs. multi ports in cell models.. 81

Table 5.1: Complexity comparison of all introduced structures.................................. 104

Table 5.2: Operation counts in two models for 100 iterations...................................... 107

Table 5.3: Execution time in seconds for lists with 10000 records. 109

Table 5.4: Execution time in seconds for lists with 50000 records. 110

Table 5.5: Execution time in seconds for lists with 100000 records. 110

Table 5.6: Execution time in micro seconds in *add* operation...................................... 111

Table 5.7: Execution time in micro seconds in *remove* operation................................ 111

Table 5.8: Execution time in micro seconds in *getMin* operation. 111

Table 5.9: Execution time in micro seconds in *getMinList* operation. 112

Table 5.10: Execution time in micro seconds in *removeMin* operation. 112

Table 6.1: Regular expressions and ports number factor in cell test. 119

Table 6.2: Regular expressions and ports number factor in block test. 119

Table 7.1: Prediction rules. 139

Table 7.2: Execution and speedup of landslide models.. 142

Table 7.3: Comparing block and cell implementation of 32×32 Cell space.................. 144

Table 7.4: The new implementation with different blocking setups 144

ABSTRACT

Recent research advances in Discrete EVent system Specification (DEVS) as well as cellular space modeling emphasized the need for high performance modeling methodologies and environments. The growing demand for cellular space models has directed researchers to use different implementation formalisms. Many efforts were dedicated to develop cellular space models in DEVS in order to employ the advantage of discrete event systems. Unfortunately, the conventional implementations degrade the performance in large scale cellular models because of the huge volume of inter-cell messages generated during simulation.

This work introduces a new multi-layer formalism for cellular DEVS models that assures high performance and ease of user specification. It starts with the parallel DEVS specification layer and derives a high performance cellular DEVS layer using the property of closure under coupling. This is done through converting the parallel DEVS into its equivalent non-modular form which involves computational and communication overhead tradeoffs. The new specification layer, in contrast to multi-component DEVS, is identical to the modular parallel DEVS in the sense of state trajectories which are updated according to the modular message passing methodology. The equivalency of the two forms is verified using simulation methods. Once the equivalency has been ensured, analysis of the models becomes a decisive factor in employing modularity in cellular DEVS models.

Non-modular models show significant speedup in simulation runs given that their event list handler is implemented based on analytical and experimental survey that involve actual operation counts. However, the new high performance non-modular specification layer is complicated to implement. Therefore, a third layer of specification is proposed to provide a simple user specification that is automatically converted into the fast complex cellular DEVS specification, which is finally put in the standard parallel DEVS specification. A tool was implemented to automatically accept user's model specification via GUI and generate the models using the new specifications. The generated models are then required to be tested and verified using some automatic DEVS verification methods. As a result, the model development and verification processes are made easier and faster.

CHAPTER 1 : INTRODUCTION

This dissertation is mainly concerned with the area of discrete event cellular model development and simulation enhancements. It introduces a new way of specifying and developing cellular DEVS models aiming simplicity, efficiency, and support of scalability.

1.1 Motivation and General Scope

The cellular space modeling approach divides space into discrete cells where local computations held in each cell are based on its own as well as its neighbor's states. In conventional DEVS implementation of cellular models (e.g. [1, 2]), a cell is implemented as a DEVS modular atomic or coupled model. When detailed modeling of spatial dynamics is required, large number of cells are typically employed. This results in a large number of atomic models that communicate through message passing to carry out the global simulation. Therefore, the task of implementing large scale cellular spaces with highly active cells in DEVS will face the burden of huge numbers of inter-cell messages and hence a performance reduction. Many techniques were introduced to resolve this issue and to gain speedup. Examples of such work can be found in [3-5] where the cellular DEVS simulation engine was improved to handle messages and cell activity scanning in more efficient manner. On the other hand, the quantized DEVS approach [6-8] shows that quantization helps in improving the performance of DEVS simulations by reducing the number of state transitions as well as the number of messages while introducing acceptable errors.

To date, research in DEVS cellular space modeling has treated each cell as an atomic or coupled model and then either sought to speed up the simulation engine or introduce quantization to the model in order to reduce messages and transitions with attendant error. The simulator enhancements were tackled by either flattening the coordinator hierarchy [5], implementing faster scheduling algorithms that deal with active cells only [3, 4], and/or eliminating unnecessary coordinator objects [3]. The work presented in this dissertation takes advantage of these enhancements and applies similar methods to the model development level. This new error-free approach is designed to reduce the number of messages by encapsulating and transforming a group of cells into non-modular form. Instead of treating a single cell as an atomic DEVS model, the encapsulation method will group a number of cells in one atomic model. The resulting model will be a non-hierarchal, non-modular cell representation that gives a significant speed up which in conjunction with the simulator enhancements done in [3-5] will give amazingly high performance on large scale distributed cellular space models over clusters.

There are very few related works that touched the area of converting coupled DEVS models into atomic models like [9] and [10]. Their implementation of the approach is by allowing the conversion during the compilation process. On the other hand, this dissertation applies the conversion into the specification and development process. The first work involved converting classical, rather than parallel DEVS, models and it did not target the cellular space in particular which made that approach a good speedup way for small size models. The other work also tried to convert large models into atomic ones to easily deal with them in Dymola. A conclusion was reached that there is no advantage of following the conversion approach since the overhead of handling large model is much greater than the messages overhead. This dissertation disagrees with that conclusion which can be considered an environment specific conclusion. Our initial work in [11] proved the significance of the approach for large cellular DEVS models.

1.2 Main Contributions

This dissertation introduced a new formalism to allow specifying the cellular DEVS models in modular as well as non-modular forms. Using the closure under coupling property of parallel DEVS, it was possible to derive a new layer of formalism to specify models in non-modular form. This new modeling layer guarantees the efficiency of the models in contrast to the current cellular DEVS implementation approaches. This was achieved by considering the non-modular form of the cellular DEVS models in which some simulator tasks were encapsulated inside the atomic cell space in order to eliminate inter-cell messages. The simulator tasks were done through event list handler which was implemented based on an analytical survey on different implementations that was also supported with experimental analysis and actual operation counts in different test models. Another layer of specification was introduced to ease and speedup the development process of complex atomic cellular DEVS models. The integrated multilayer formalism, introduced in this work, supports the automation of model development and transformation between the different layers.

A new cellular DEVS development environment was introduced and implemented as an outcome of this work. This environment supports the multilayer development approach since it was built to run over general DEVSJAVA simulators. The automated processes introduced in this tool are: automated specification transformation, automated code generation, and automated testing and model verification. This environment makes it possible for the first time to develop cellular DEVS models using GUI without writing the full DEVSJAVA code. It was made as generic as possible to eliminate the need for the user to modify the code generated by the environment. In addition, some automated testing classes were also made available for the user in order to test and verify the developed models.

1.3 Dissertation Outline

The remaining of the dissertation can be outlined as follows. Chapter 2 gives a quick background on parallel DEVS, cellular DEVS and all necessary theory required as a basis of this dissertation. Then, Chapter 3 introduces the new high performance cellular DEVS formalism that was derived based on the parallel DEVS formalism and the closure under coupling property. All equations used to formulate the new specification are presented in addition to a set of lemmas that prove the support of the new approach to a wide variety of cellular models. Chapter 4 presents the new development environment as well as its design aspects. It first explains the user specification layer that is used to design the graphical user interface. The major critical design issue in the new environment is the event list handler implementation which is discussed in Chapter 5 with more details, discussions, experiments and analysis. The next chapter is devoted to describe the procedures and the proposed methods followed to test, validate and verify our modeling tool as well as its generated models. It involves the techniques used in this work to verify the new approach and ensure its equivalency to the conventional cellular DEVS approaches. Chapter 7 illustrates the capability of our environment to support complex cellular DEVS models through implementing and modifying different landslide models. It also shows the significance of our new approach through presenting the simulation runs of the landslide model. Finally, the last chapter concludes this dissertation and overview some future works.

CHAPTER 2 : BACKGROUND

2.1 Cellular Automata

Cellular Automata (CA), which was first introduced by John Von Neumann in the 1950s [12], has been widely used in simulating complex systems. The domain of application of CA includes fluid and mass flow [13-15], natural hazards [16], many other sorts of pattern recognition [17, 18], image processing [19], ecosystems [20], and traffic modeling [21, 22]. In addition, it has been used as solutions for common computational needs like networking [23], solving differential equations [24], and distributed computing [25-28].

CA is a discrete time dynamical system that consists of a lattice of cells in single or multi-dimension in which each cell applies a local transition function to calculate its next state [29]. Many efforts in recent years were dedicated to improve simulation methodologies that take advantage of CA in modeling complex behavioral dynamic systems (e.g., see [12, 14]). Based on the fixed time step CA, Avolio and his coworkers developed an empirical approach for modeling and simulating complex dynamic systems [14]. Such an approach can be applied to problems that are very difficult to manage with differential equations systems. However, the use of discrete events, rather than fixed time steps, in simulation gives a significant speedup in many applications [6, 30-32]. As a result, DEVS (Discrete EVent System Specification) has been attracting many researchers as a basis of CA modeling of complex physical systems. Its parallel version, parallel DEVS, was introduced in [33, 34] to provide a sound framework that exploit the parallelism of the hierarchical DEVS models. Based on this parallelism, it was possible to introduce Cellular DEVS [6] and Cell-DEVS [35, 36] which integrate the theories and algorithms of CA in DEVS.

2.2 Parallel DEVS Formalism (P-DEVS)

Discrete EVent System Specification (DEVS) [6] supports object orientation over modeling environments. Its theory provides a mathematical formalism for representing dynamic systems. The DEVS formalism was revised in [33] to reduce sequential processing and enable full parallel executions. The resulting parallel DEVS has the basic atomic model defined as:

$$M = \left\langle X, S, Y, \delta_{\mathrm{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \right\rangle,$$

where

X is a set of input values (set of ports/values in coupled structures)

S is a set of states (set of ports/values in coupled structures)

Y is a set of output values

δ_{int}: $S \to S$ is the *internal transition* function

δ_{ext}: $Q \times X^b \to S$ is the *external transition* function

$Q = \{(s,e) \mid s \in S, 0 \le e \le ta(s)\}$ is the *total state* set

e is the *time elapsed* since last transition

X^b denotes the collection of bags over X

δ_{con}: $S \times X^b \to S$ is the *confluent transition* function

λ: $S \to Y^b$ is the output function *ta*: $S \rightarrow R_{0\rightarrow\infty}$ is the time advance function.

An atomic model M in parallel DEVS remains in a state $s \in S$ for $ta(s)$ amount of time if no external event occurs. When that time advance expires, i.e., when the elapsed time, $e = ta(s)$, the system outputs the values, $Y^b = \lambda(s)$, just before it changes to state $\delta_{int}(s)$. When an external event x in X^b occurs before this expiration time, i.e., at $e < ta(s)$, the system changes to state $\delta_{ext}(s,e,x)$. However, in case of internal and external transitions collide, δ_{con} is employed to resolve the conflict and determine the next state. In all cases, the model then goes to some new state s′ with some new resting time, $ta(s')$ and the same story continues [6].

Note that input or output values X^b and Y^b are bags of elements. This means that one or more elements can appear on a port at the same time. This capability comes from the parallel implementation of DEVS which allow components to send to the ports simultaneously. These basic components may be coupled in DEVS to form a multi-component model which is defined by the following structure:

$$CM = \left\langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \right\rangle,$$

where

X is the set of input values (set of ports/values in coupled structures)

Y is the set of output values (set of ports/values in coupled structures)

D is the set of components

for each i in D: M_i is a component which is an atomic model M_i = $\langle X_i, S_i, Y_i, \delta_{\text{int}\,i}, \delta_{ext\,i}, \delta_{con\,i}, \lambda_i, ta_i \rangle$

for each i in $D \cup \{\text{self}\}$: I_i is the influencees of i, i is not in I_i

self is the coupled model itself *CM* which allow external inputs and outputs

for each j in I_i : $Z_{i,j}$ is the i to j output translation function (coupling)

$Z_{self,j} : X_{self} \rightarrow X_j$

$Z_{i,self} : Y_i \rightarrow Y_{self}$

$Z_{i,j} : Y_i \rightarrow X_j$

2.2.1 Closure under Coupling of Parallel DEVS

Closure under coupling, in parallel DEVS, states that every coupled model (*CM)* has its own equivalent atomic model (*M*). This section demonstrates the closure derivation in order to reach the resultant atomic model of any coupled model. Generally speaking, the coupled DEVS model can be treated as a black box with input as well as output ports (*X* and *Y*) that form the first terms of equivalency in the atomic and coupled models. The state set (*S*) of the resultant model will be the total state sets of all the atomic coupled models. In addition, the time advance *ta(s)* will be the minimum of all the internal atomic models.

$$S = \underset{d \in D}{\times} Q_d \text{, where } s \in S \text{, and } s = (\ldots, (s_d, e_d), \ldots) \text{ for all } d \in D$$

$$ta(s) = minimum\{\sigma_d \mid d \in D\}, \text{ where } s \in S \text{ and } \sigma_d = ta(s_d) - e_d$$

We define different sets of internal components according to their states during any iteration (at a given global state s) during the simulation:

$IMM(s)=\{d \mid \sigma_d=ta(s)\}$	*Set of imminent components*
$INF(s)=\{d \mid i\in I_d,\ i\in IMM(s) \wedge x_d^b \neq \emptyset\}$,	*Set of components about to receive inputs*
	where $x_d^b=\{Z_{i,d}(\lambda_i(s_i)) \mid i\in IMM(s)\cap I_d\}$
$CONF(s)=IMM(s)\cap INF(s)$	*Set of confluent components*
$INT(s)=IMM(s)-INF(s)$	*Set of imminents those receiving no inputs*
$EXT(s)=INF(s)-IMM(s)$	*Set of non-imminents those receiving input*
$UN(s)=D-IMM(s)-EXT(s)$	*Other components*

Based on these groups we just need to formulate the functions (λ, δ_{int}, δ_{ext}, δ_{con}) for the resultant model as follows:

$\lambda(s)=\{Z_{d,self}(\lambda_d(s_d)) \mid d\in IMM(s) \wedge d\in I_{self}\}$,

$\delta_{int}(s) = (\ldots, (s_d', e_d'), \ldots)$,

$$\text{where } (s'_d, e'_d) = \begin{cases} (\delta_{\text{int},d}(s_d), 0) & d \in INT(s) \\ (\delta_{ext,d}(s_d, e_d + ta(s), x_d^b), 0) & d \in EXT(s) \\ (\delta_{con,d}(s_d, x_d^b), 0) & d \in CONF(s) \\ (s_d, e_d + ta(s)) & otherwise \end{cases},$$

$\delta_{ext}(s,e,x^b) = (\ldots, (s_d', e_d'), \ldots)$,

$$where\ 0<e<ta(s)\ and\ (s'_d, e'_d) = \begin{cases} (\delta_{ext,d}(s_d, e_d + e, x_d^b), 0) & self \in I_d \wedge x_d^b \neq \Phi \\ (s_d, e_d + e) & otherwise \end{cases},$$

and $x_d^b = \{Z_{self,d}(x) \mid x\in x^b \wedge self\in I_d\}$.

For the last function δ_{con} we need to redefine the group *INF(s)* to include the set of influencees by external events to the whole model. Let $INF'(s) = \{d|(i\in I_d,\ i\in IMM(s) \vee self\in I_d) \wedge x_d^b \neq \varnothing\}$, where $x_d^b = \{\{Z_{i,d}(\lambda_i(s_i))|\ i\in IMM(s) \wedge i\in I_d\} \cup \{Z_{self,d}(x)|\ x\in x^b \wedge self\in I_d\}\}$. Then, we need to redefine the other groups accordingly:

$CONF'(s)=IMM(s)\cap INF'(s)$

$INT'(s)=IMM(s)-INF'(s)$

$EXT'(s)=INF'(s)-IMM(s)$

As a result,

$\delta_{con}(s,x^b) = (\ldots, (s_d',e_d'), \ldots),$

$$where\ (s'_d, e'_d)\begin{cases} (\delta_{\text{int},d}(s_d),0) & d\in INT'(s) \\ (\delta_{ext,d}(s_d, e_d + ta(s), x_d^b),0) & d\in EXT'(s) \\ (\delta_{con,d}(s_d, x_d^b),0) & d\in CONF'(s) \\ (s_d, e_d + ta(s)) & otherwise \end{cases}.$$

The over all transition function will be:

$$\delta(s,e,x^b) = \begin{cases} \delta_{ext}(s,e,x^b) & 0\le e < ta(s)\wedge x^b \neq \Phi \\ \delta_{con}(s,x^b) & e = ta(s)\wedge x^b \neq \Phi \\ \delta_{\text{int}}(s) & e = ta(s)\wedge x^b = \Phi \end{cases}$$

2.3 Multi-Component DEVS Formalism

The previous sections represent the parallel DEVS formalism in its atomic as well as its coupled and hierarchical forms. All these mentioned specifications are in the modular form in which components have no means of accessing other component's states and variables except through ports and messages. The other form is the non-modular form which is referred to as multi-component DEVS in [6]. In multi-component DEVS, which is based on classical DEVS, components of coupled models can directly influence each other through their state transitions. That means events occurring in one component may result in state changes and rescheduling of events in other components. A Multi-component DEVS can be defined as follows [6]:

$$\text{multiDEVS} = \langle X, Y, D, \{M_d\}, Select \rangle,$$

where

X, Y are the input and output event sets

D is the set of component references

$Select: 2^D \rightarrow D$ with $Select(E) \in E$ is a tie-breaking function employed to arbitrate in case of simultaneous events.

For each $d \in D$

$M_d = \langle S_d, I_d, E_d, \delta_{ext,d}, \delta_{int,d}, \lambda_d, ta_d \rangle$

where

S_d is the set of sequential states of d,

$Q_d = \{(s, e_d) \mid s \in S_d, e_d \in \Re\}$ is the set of total states of d,

$I_d \subseteq D$ is the set of influencing components,

$E_d \subseteq D$ is the set of influenced components,

$\delta_{ext,d}: \times_{i \in Id} Q_i \times X \rightarrow \times_{j \in Ed} Q_j$ is the external state transition function,

$\delta_{int,d}: \times_{i \in Id} Q_i \rightarrow \times_{j \in Ed} Q_j$ is the internal state transition function,

$\lambda_d: \times_{i \in Id} Q_i \rightarrow Y$ is the output event function, and

ta_d: $\times_{i \in Id} Q_i \rightarrow \Re^{+}_{0} \cup \{\infty\}$ is the time advance function.

Any component $d \in D$ in multiDEVS can schedule its own internal event with its own time advance ta_d. On event occurrence, in this component, its internal transition function $\delta_{int,d}$ is executed to generate a state transition as well as an output event defined by λ_d. The transition function depends on total states q_i of the influencing components I_d and changes any total state q_j of the influenced components E_d. The *Select* function is used as a tie breaking function that selects the component to be executed in the case where different components are imminent. Any external events received by the multiDEVS's ports will be handled by the corresponding component that should receive that specific event through $\delta_{ext,d}$. However, $\delta_{ext,d}$ can be left undefined for components which are not needed to receive input events and, similarly, there is no need to define λ_d for the components which are not expected to send outputs.

2.4 Cellular Space Models in DEVS

The cellular automata applications, based on the discrete time simulation, consume the computation power in doing computations to update all cells in every single iteration. In a wide range of applications, there are a lot of cells that are not required to be updated at every step which makes the discrete time approach inefficient. In addition, the selection of the time step size has a significant impact on the simulation accuracy. High accuracy requires a very small step size which, in turn, requires huge computational resources. The discrete event approach overcomes these problems by dedicating computational resources to the cells that actually perform state transitions and hence avoiding unnecessary computation on inactive cells. Due to these advantages, many efforts were dedicated to employ the DEVS approach to cellular automata applications (e.g. [1, 5, 37, 38]).

The conventional cellular DEVS approaches divide the spatial space into discrete cells where local computations are done in each cell. A cell is implemented as an atomic DEVS model which performs the local computations internally based on its own state as well as the neighboring states that are received through the external ports. The cell space is implemented as a coupled DEVS model that contains a number of cells that are arranged in an array. The neighboring rule followed in a specific application determines the internal port couplings between cells and the boundary couplings that connect the cells at the borders with other cells in different cell spaces. Figure 2.1 illustrates the conventional 2-D cellular space implementation in the DEVS formalism.

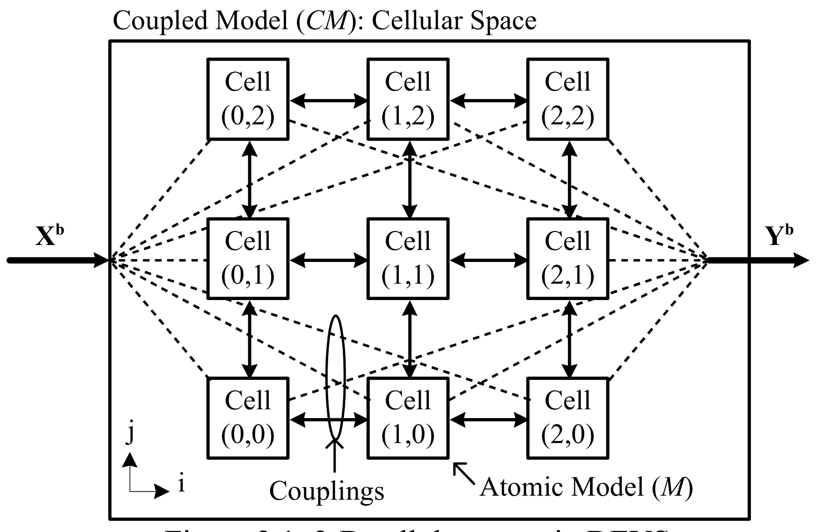


Figure 2.1: 2-D cellular space in DEVS.

2.4.1 Closure under Coupling for Cellular Models in Parallel DEVS

Cellular space models are characterized by identical cells that are spatially distributed over a given area and each cell applies the same transition as well as output functions to the data and states of the area it covers. Given a coupled model *CM* = $\left\langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \right\rangle$ representing the total cellular space, each cell will be an atomic DEVS model presented as a structure $\left\langle X^*, S^*, Y^*, \delta^*_{\text{int}}, \delta^*_{ext}, \delta^*_{con}, \lambda^*, ta^* \right\rangle$. Following the same formulation of the closure under coupling of parallel DEVS, we will end up with the same formulas as above with

$\delta_{int,\,d} = \delta^*_{\text{int}}$, $\delta_{ext,\,d} = \delta^*_{ext}$, $\delta_{con,\,d} = \delta^*_{con}$, $\lambda = \lambda^*$, $ta_d(s) = ta^*(s)$ for all $d \in D$

The resultant atomic model of the complete cellular space will be:

CM= $\left\langle X, S, Y, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \right\rangle$

with the following functions:

$\lambda(s) = \{ Z_{d,self}(\lambda^*(s_d)) \mid d \in IMM(s) \wedge d \in I_{self} \}$

$\delta_{int}(s) = (\ldots, (s_d', e_d'), \ldots)$

$$\text{where } (s'_d, e'_d) = \begin{cases} (\delta^*_{\text{int}}(s_d), 0) & d \in INT(s) \\ (\delta^*_{ext}(s_d, e_d + ta(s), x^b_d), 0) & d \in EXT(s) \\ (\delta^*_{con}(s_d, x^b_d), 0) & d \in CONF(s) \\ (s_d, e_d + ta(s)) & otherwise \end{cases}$$

$\delta_{ext}(s, e, x^b) = (\ldots, (s_d', e_d'), \ldots),$

$$where\ 0<e<ta(s)\ and\ (s'_d, e'_d) = \begin{cases} (\delta^*_{ext}(s_d, e_d + e, x^b_d), 0) & self \in I_d \wedge x^b_d \neq \Phi \\ (s_d, e_d + e) & otherwise \end{cases}$$

$\delta_{con}(s, x^b) = (\ldots, (s_d', e_d'), \ldots),$

$$where\ (s'_d, e'_d) = \begin{cases} (\delta^*_{\text{int}}(s_d), 0) & d \in INT'(s) \\ (\delta^*_{ext}(s_d, e_d + ta(s), x^b_d), 0) & d \in EXT'(s) \\ (\delta^*_{con}(s_d, x^b_d), 0) & d \in CONF'(s) \\ (s_d, e_d + ta(s)) & otherwise \end{cases}$$

Similarly, the over all transition function will be:

$$\delta(s, e, x^b) = \begin{cases} \delta_{ext}(s, e, x^b) & 0 \le e < ta(s) \wedge x^b \ne \Phi \\ \delta_{con}(s, x^b) & e = ta(s) \wedge x^b \ne \Phi \\ \delta_{\text{int}}(s) & e = ta(s) \wedge x^b = \Phi \\ (s, e) & otherwise \end{cases}$$

2.5 Related Work

2.5.1 Cell-DEVS Formalism

Cell-DEVS formalism was introduced to employ the advantages of discrete event systems in cellular automata applications [36]. It is an extension to the DEVS formalism that makes the cell timing specification more expressive [39]. This was achieved by adding more entries to the original DEVS atomic model specification in order to specify cells with local computing function and transport as well as inertial delays. Cell-DEVS atomic model is specified as:

$$\text{CD} = \langle X, Y, I, S, N, delay, d, \delta_{\text{int}}, \delta_{ext}, \tau, \lambda, D \rangle$$

The following terms are defined in similar way as in the standard DEVS atomic models: X, Y, S, δ_{int}, δ_{ext}, and λ. All other terms are defined as follows: D is the time advance spent in a state which is referred to as *ta* in DEVS, *delay* is the type of cell's delay, *d* is the duration of that delay, *I* is the cell's modular Interface, *N* is the set of input events, and τ is the local computation function. In addition to the atomic DEVS operations, each cell, in Cell-DEVS, receives the set of *N* inputs through the model interface *I* that activate the local computation function through δ_{ext}. There are two types of delays were introduced: the transport and the inertial delay with a specified duration *d* that plays role in determining the actual time to execute the scheduled event and to send the output messages.

The cell space model in Cell-DEVS is a coupled DEVS model that contains a number of atomic cells that are interconnected through ports following some neighboring rules. The cells are arranged in a single or multi-dimensional array that is coupled at borders to allow connecting the cell space into other spaces within a global multi-space model [39]. Cell-DEVS formalism, since it represents each cell as an atomic model, is considered as a conventional DEVS implementation of cell space models which has the performance drawback that is resulted by the huge volume of inter-cell communication generated during simulation. In addition, expressing cellular models in Cell-DEVS formalism is, to some extent, complex and requires more efforts at the modeler level. On the other hand, this dissertation introduces the multi-layer approach to simplify the modeling process and make the cell space's extensive specifications transparent to the end user.

2.5.2 Converting Coupled Model into Atomic DEVS

Closure under coupling property proved that any DEVS coupled model can be represented in an atomic DEVS form. The proof followed in the previous sections did not completely decompose the internal atomic models. It just obtained the basic atomic functions for the overall coupled model as a black box where the internal models still keep the same DEVS atomic structures in modular form. However, this property initiated the idea of converting coupled models into atomic models for the purpose of simulation speedup.

Lee and Kim [9] introduced a composition-based method that converts a coupled classical DEVS model into atomic classical DEVS model at compile time. Their goal was to achieve simulation speedup by computing possible event and message routes at compile time which are then handled by a simulation engine inside the composed atomic model. The formal approach, they presented, followed the same idea of the closure under coupling formulation with addition of a scheduling mechanism. Therefore, it still keeps the DEVS atomic structure of the internal models and if not, the resultant atomic model should keep track of all the functions of these internal models. This will introduce an overhead where the model will be required at each event processing to search the whole pool of functions to get the corresponding function for that event and as the number of the internal models grows very large, this overhead will be a bottle neck in achieving a good speedup in large scale coupled models. Furthermore, the conversion process will be more complicated to be done correctly. These drawbacks were the reason behind what Beltrame [10] concluded by following the same approach. In that thesis, the idea of converting a coupled model into atomic one was implemented in order to eliminate the message overhead by using Modelica's parallel variable update. Instead of gaining speedup, the models show slowness in simulation runs because of the large amount of variables and functions bookkeeping.

CHAPTER 3 : NEW FRAMEWORK FOR CELLULAR DEVS MODELING

In this chapter we formulate a new cellular space DEVS specification to achieve a fast cell space model development process as well as a fast simulation execution. The basic idea is to divide the cell space into blocks, each with a fixed number of cells and convert these blocks entirely into DEVS atomic models. This process can be seen as converting modular cells into non-modular form inside the block where each cell can access the state variables of its neighboring cells. Based on the closure under coupling property of the parallel DEVS, the conversion process is known to be complex for the end user which, in turn, requires us to consider ease of user specification in the resulted framework.

3.1 Converting Cell Space Model into Atomic DEVS

As a special case of the closure under coupling property in DEVS, cellular space models can take advantage of this property in gaining speedup by converting coupled models into atomic ones. In this approach, scalability will not be an issue since all cells (i.e. atomic models) have identical transition functions that will be applied to all cells iteratively. This implies that the model will not be required to search for the functions in a huge pool of functions for the encapsulated atomic models. In addition, by fully decomposing the internal models into non-modular form, the inter-cell communication messages will be eliminated and result in simulation speedup.

3.2 Toward Full Decomposition of Cell Space Models

The following sections describe the process of formulating the atomic model specification of the coupled cell space models. The aim behind this is to get simulation speed up in large scale complex cellular models. In contrast to the modular techniques, the non-modular ones show great speedups since components can access other component's states directly with no inter-component messages. The following procedure is based on the modular closure under coupling of parallel DEVS since most of the recently implemented DEVS environments are in the modular parallel form. Starting from that form, we propose some speed up techniques that will be based on converting the internal atomic models into non-modular form.

3.3 Closure under Coupling of Parallel DEVS Applied to Cell Spaces

Closure under coupling of parallel DEVS states that a coupled model can be represented with its atomic P-DEVS equivalent which is defined as follows:

Given P-DEVS coupled model $\left\langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \right\rangle$, we define a basic model $\left\langle X^a, S, Y^a, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \right\rangle$.

Where $M_i = \left\langle X_i, S_i, Y_i, \delta_{\text{int}\,i}, \delta_{ext\,i}, \delta_{con\,i}, \lambda_i, ta_i \right\rangle$

for each $i \in D$, $X \equiv X^a$, $Y \equiv Y^a$, $S = \underset{d \in D}{\times} Q_d$, $ta(s) = minimum\{\sigma_d \mid d \in D\}$, $s \in S$, $\sigma_d = ta(s_d) - e_d$, and the transition functions are defined as follows:

δ_{int}: $\underset{d \in D}{\times} Q_d \rightarrow \underset{d \in D}{\times} Q_d$

δ_{ext}: $\underset{d \in D}{\times} Q_d \times X \rightarrow \underset{d \in D}{\times} Q_d$

δ_{con}: $\underset{d \in D}{\times} Q_d \times X \rightarrow \underset{d \in D}{\times} Q_d$

λ: $\underset{d \in D}{\times} Q_d \rightarrow Y$

In the case of cellular space models, all components $d \in D$ are P-DEVS atomic cells which are identical processing objects with the same transition functions as well as output functions $\delta^*_{\text{int}}, \delta^*_{ext}$, δ^*_{con} and λ^*. For this special case of the DEVS coupled model, the resultant overall cell space atomic functions will be as follows:

$\lambda(s) = \{ Z_{d,self}(\lambda^*(s_d)) \mid d \in IMM(s) \wedge d \in I_{self} \}$

$\delta_{int}(s) = (\ldots, (s_d', e_d'), \ldots)$

$$\text{where } (s'_d, e'_d) = \begin{cases} (\delta^*_{\text{int}}(s_d), 0) & d \in INT(s) \\ (\delta^*_{ext}(s_d, e_d + ta(s), x^b_d), 0) & d \in EXT(s) \\ (\delta^*_{con}(s_d, x^b_d), 0) & d \in CONF(s) \\ (s_d, e_d + ta(s)) & otherwise \end{cases}$$

$\delta_{ext}(s, e, x^b) = (\ldots, (s_d', e_d'), \ldots)$

$$where\ 0<e<ta(s)\ and\ (s'_d, e'_d) = \begin{cases} (\delta^*_{ext}(s_d, e_d + e, x^b_d), 0) & self \in I_d \wedge x^b_d \neq \Phi \\ (s_d, e_d + e) & otherwise \end{cases}$$

$\delta_{con}(s, x^b) = (\ldots, (s_d', e_d'), \ldots)$

$$where\ (s'_d, e'_d) = \begin{cases} (\delta^*_{\text{int}}(s_d), 0) & d \in INT'(s) \\ (\delta^*_{ext}(s_d, e_d + ta(s), x^b_d), 0) & d \in EXT'(s) \\ (\delta^*_{con}(s_d, x^b_d), 0) & d \in CONF'(s) \\ (s_d, e_d + ta(s)) & otherwise \end{cases}$$

3.3.1 Event List Handling

The approaches mentioned in section 2.5.2 used the idea of closure under coupling to decompose general (non cell space) coupled models to gain some speed up. Unfortunately, those approaches do not guarantee speed up in large scale models. The first factor of scalable speedup in our approach is that it targets cell space models where all the cells are having the same transition functions. This property will ease the process of decomposing coupled cell spaces by moving these transition functions to the cell space level and implementing an iterative approach to apply these functions to the active cells. Since we are implementing the whole framework in the discrete event simulation domain, we need a discrete event list handler that manages, for the cell space, the list of active cells at each simulation time, namely the cells to which the cell transition functions must be applied.

Introducing the event list obviates the requirement that each cell keeps track of its own timing since time management will be handled by scheduling events on the events list. The events list is employed at the level of the resultant atomic model which will contain the future events expected to happen for the cells. An event record will be in the form that describes when the event is scheduled to occur (next event time) and where it will happen (which cell). The elapsed time of each cell inside the DEVS coupled model is normally handled by the coordinator. However, since we are replacing the coupled model by its atomic model equivalent, it will be the atomic model's responsibility to keep records of the elapsed time for each of its internal cells. Therefore, the atomic model should keep a variable that store the current simulation time for the cell space block that it represents and a vector of cell's history times which store when a specific cell was accessed or had a transition time last. The elapsed time can be obtained by subtracting the history time of a specific cell from the current simulation time.

Within the DEVS framework, a cell can only receive an external message when the time advance has expired at another cell inside the same coupled model or an external message was received by the coupled model's input ports. Accordingly, the events list implementation just stores the time advances of the active cells and upon time advance expirations, the event list handler must add the neighboring cells of the imminent cells to a receiver group. This group will be the list of the cells that may receive external messages. In addition, when the coupled model receives external messages, the handler should identify the cells that should be aware of these new messages and add them to the scan list for external transitions. The events list handler is the responsibility of the resultant atomic model which, as a DEVS model, must implement all functionality solely using its transition functions.

Now, we have a cell space having an *EVENTS* list with events *ev = (time, i),* where

$ev \in EVENTS,\ time: R_{0\to\infty},\ and\ i \in D.$

Therefore,

$$CellSpace = \langle X, Y, D, \{Cell_i\}, \{I_i\}, \{Z_{i,j}\}, EVENTS \rangle \rightarrow \langle X, S, Y, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

where

$Cell_i = \langle X_i, S_i, Y_i \rangle$ *for each* $i \in D$,

$ta(s) = minimum\ \{time\ |\ (time, i) \in EVENTS\}.$

This means that in the new representation, the cell is no longer an active processing unit. It just stores the state variables with no timing involved at its level. It still has ports and messages which means it is not yet in non-modular form. The task of conversion to non-modular form will be done in the next subsection.

Now, the events list will play role in defining cell groups as follows:

$IMM(s)=\{ j \mid (ta(s), j) \in EVENTS \}$

Given that $X^b{}_j=\{Z_{i,j}(\lambda^*(s_i)) \mid i \in IMM \wedge i \in I_j\}$ *, the collected outputs of the imminents*

$INF=\{ j \mid (i \in I_j \mid ((ta(s),i) \in EVENTS) \wedge X^b{}_j \neq \emptyset)\}$*, receiving cells influencees*

$INT =\{ j \mid ((ta(s), j) \in EVENTS \wedge X^b{}_j = \emptyset)\}$,

$EXT=\{ j \mid (i \in I_j \mid (ta(s),i) \in EVENTS \wedge (ta(s),j) \notin EVENTS \wedge X^b{}_j \neq \emptyset)\}$

$CONF= \{ j \mid ((ta(s), j) \in EVENTS \wedge X^b{}_j \neq \emptyset)\}$

In addition, given that $X^b{}_j=\{Z_{i,j}(\lambda^*(s_i)) \mid (i \in IMM \wedge i \in I_j)\} \cup \{Z_{self,j}(x) \mid x \in x^b \wedge self \in I_j \}$

$INF'=\{ j \mid ((i \in I_j \mid (ta(s),i) \in EVENTS) \vee (self \in I_j)) \wedge X^b{}_j \neq \emptyset)\}$

$INT'= \{ j \mid ((ta(s), j) \in EVENTS \wedge X^b{}_j = \emptyset)\}$

$CONF'= \{ j \mid ((ta(s), j) \in EVENTS \wedge X^b{}_j \neq \emptyset)\}$

$EXT'=\{ j \mid ((i \in I_j \mid (ta(s),i) \in EVENTS) \vee (self \in I_j)) \wedge (ta(s),j) \notin EVENTS \wedge X^b{}_j \neq \emptyset)\}$

According to these new definitions, we can reformulate the resultant transition functions of the atomic cell space to include the events list handling. The following formal details show how the cell groups are extracted from the event list and manipulated in the atomic transition functions. The only function that does not deal with these cell groups extracted from *EVENTS* list, is the external function. It just deals with the cells that received external inputs through the cell space ports.

At the end of every transition cycle of the atomic cell space, the model checks the events list to see if it contains more scheduled events and if so, it extracts the list of cells with minimum time advance. On the expiration of that minimum time advance, the output function will access the *IMM* list and let the cells in this group send their output messages. Then, the internal or confluent transition functions are responsible to obtain the corresponding cell groups where the external transition function will work on the boundary cells that received external messages. Another source of speed up can be achieved here by letting the events list only hold non-infinity time advances which is equivalent to enhancing the simulator. This means that the model will not waste time by dealing with passive cells which is the case in any DEVS simulator when each cell is implemented as an atomic model.

```
$\lambda(s)$ :        for all $i \in IMM$
                          Apply $\lambda^*$ to $Cell_i$   [ $Y_i=\lambda^*(s_i)$ ]
$\delta_{int}(s)$ :     obtain cell groups INT, EXT, CONF
                      Update EVENTS:     time=time-ta(s) for all ( time, i) $\in$ EVENTS
                                         delete any ev=( 0 , i ) where ev $\in$ EVENTS
                      for all $i \in INT$:     apply $\delta^*_{int}$ to $Cell_i$
                                         schedule (next ta, i)
                      for all $i \in EXT$:     apply $\delta^*_{ext}$ to $Cell_i$
```

```
                              schedule (next ta, i)
            for all $i \in CONF$:    apply $\delta^*_{con}$ to $Cell_i$
                              schedule (next ta, i)
            if (EVENTS $\neq$ {})
                extract IMM from EVENTS
                ta(s) = minimum {time |( time, i) $\in$ EVENTS }
            else
                clear IMM={}
                ta(s) = $\infty$
$\delta_{con}(s,x^b)$ :   obtain cell groups INT', EXT', CONF'
            Update EVENTS:   time=time-ta(s) for all ( time, i) $\in$ EVENTS
                              delete any ev=( 0 , i ) where ev $\in$ EVENTS
            for all $i \in INT'$ :   apply $\delta^*_{\text{int}}$ to $Cell_i$
                              schedule (next ta, i)
            for all $i \in EXT'$ :   apply $\delta^*_{ext}$ to $Cell_i$
                              schedule (next ta, i)
            for all $i \in CONF'$ :  apply $\delta^*_{con}$ to $Cell_i$
                              schedule (next ta, i)
            if (EVENTS $\neq$ {})
                extract IMM from EVENTS
                ta(s) = minimum {time |( time, i) $\in$ EVENTS }
            else
                clear IMM={}
                ta(s) = ∞
$\delta_{ext}(s,e,x^b)$ :    Update EVENTS:    time=time-e for all ( time, i) $\in$ EVENTS
            for all i $\in$ { j | self $\in I_j \wedge X^b{}_j \neq \emptyset$}
                                apply $\delta^*_{ext}$ to $Cell_i$
                                schedule (next ta, i)
            if (EVENTS $\neq$ {})
                extract IMM from EVENTS
                ta(s) = minimum {time |( time, i) $\in$ EVENTS }
            else
                clear IMM={}
                ta(s) = ∞
```

3.3.2 Transforming Cells to Non-Modular Form

So far, all the above specifications are in the modular form, which means that there are still number of internal messages passing between the internal cells in order to know the states of each other through the output functions and ports. A major additional speed up can be achieved by transforming these cells into non-modular form. In non-modular form, a cell can access (read) the state variables of its neighbors and there is no need for message passing through ports. In contrast to multi-component DEVS [6], the implementation we are seeking here allows cells to read each other's states, but they are only allowed to make their own state transitions. In case a cell changes its state, its neighboring cells need to be added to the cell group *EXT* instead of allowing the cell itself to change their states directly.

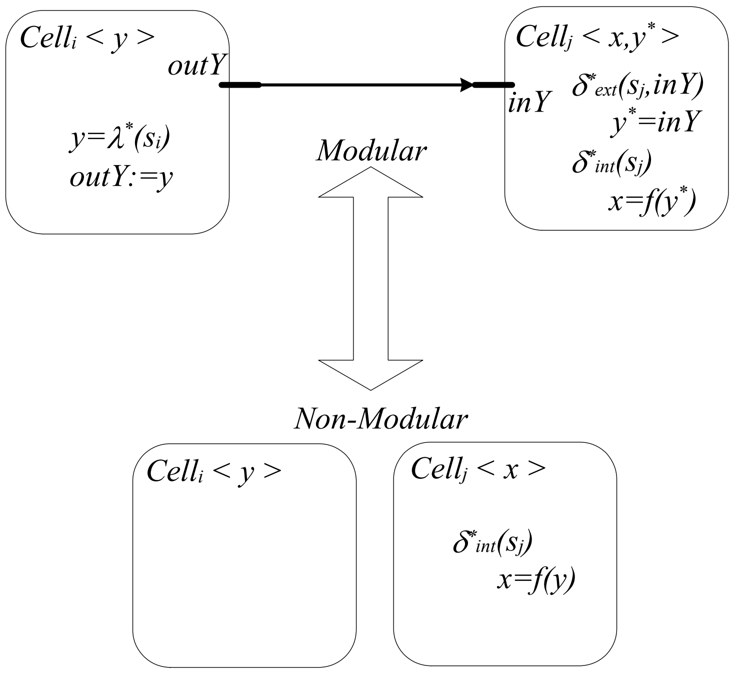


Figure 3.1: Switching between modular and non-modular forms.

Figure 3.1 shows how models can be transformed from modular into non-modular form and vice versa. Transforming to non-modular form can be achieved by removing the ports from the atomic models and letting them directly access the neighboring models to read their state variables. In cellular space models, this will reduce the structure of the cell units into a smaller one that just stores states and variables only and it has no functions or processing done at its level. However, we need to keep the coupling relations so that each cell knows which neighboring cells to access. In this case, we can add to the cell structure a list of the neighboring cells (*n*) which can be accessed by that specific cell. In cellular space models, this list is the set of influencers which is equivalent to the set influencees.

$Cell_i = \langle X_i, S_i, Y_i \rangle \rightarrow Cell_i = \langle S_i, n_i \rangle$, *Where* $n_i = \{ j \mid j \in I_i \} = \{ j \mid i \in I_j \}$

Since we removed the ports from the cells, the cell space atomic model functions need to be redefined according to the new changes. The first change is that we do not need cells to generate outputs to ports since their neighboring cells can access their state variables directly. Therefore, when a cell goes through a state transition, its neighboring cells are added to the set of cells, *EXT*, that should fire their external transition functions.

Assumption-1:

The output values that are sent via messages by a cell in the modular form are actually values of one or more of its state variables.

Applying this assumption requires that we first define the state variables of each cell as follows: Given that each cell has a state set S_i, and each state $s_i \in S_i$ is a collection of values of the state variables that represent the current state of the cell *i*. Therefore, $s_i=(sv^1_i, sv^2_i, sv^3_i, \ldots sv^n_i)$ where *n* is the number of state variables in the cell. Note that the primary states (e.g. passive, active … etc) are also treated as one of the state variables which contain the name of the state as a string.

The resultant non-modular cell space can be shown to be equivalent to the modular counterpart as follows: Given that assumption-1 is satisfied, each modular cell *i* will send its own values of the state variables through Y^b_i to its neighbors whenever there is a change in those values. Then, that neighboring cell *j* receives those values at X^b_j. Therefore, $Y^b_{i,j} = X^b_{i,j}$ for all cells *i,j* ∈*D* and there exist coupling relation $Z_{i,j}$. However, the initiating cell, *i,* actually sends its own selected set (v_{ij}) of state values to neighbor *j* and so, $Y^b_{i,j} = \{sv^k_i \mid (k \in v_{ij}) \wedge (1 \leq k \leq n)\}$ which represent the set of state values that should be sent to cell *j*.

Then, given that there exist the coupling $Z_{i,j}$ where $i,j \neq self$,

$$X^b_{i,j} = Y^b_{i,j} = \{sv^k_i \mid (k \in v_{ij}) \wedge (1 \leq k \leq n)\} ,$$

$$Y^b_i = \underset{i \in I_j}{\times} Y^b_{i,j} = \underset{i \in I_j}{\times} \{sv^k_i \mid (k \in v_{ij}) \wedge (1 \leq k \leq n)\} = \{sv^{k11}_i, sv^{k12}_i, sv^{k13}_i, ..., sv^{k21}_i, sv^{k22}_i, sv^{k23}_i, ...\}$$

with $kj1, kj2, kj3, ... \in v_{ij}$, and

$$X^b_j = \underset{i \in I_j}{\times} X^b_{i,j} = \underset{i \in I_j}{\times} \{sv^k_i \mid (k \in vij) \wedge (1 \leq k \leq n)\} = \{sv^{k11}_{i1}, sv^{k12}_{i1}, sv^{k13}_{i1}, ..., sv^{k21}_{i2}, sv^{k22}_{i2}, sv^{k23}_{i2}, ...\},$$

where $ki1, ki2, ki3, ... \in v_{ij}$.

That means that all input/output values are equivalent to the state variables of the cells. Therefore, we can redesign the cell space model to make it fully non-modular by making each cell access the required state variables of its neighboring cells while still keeping the equivalency given that assumption-1 is met. According to Figure 3.1, implementing modular cells makes each cell keep records of its neighboring state variable (y^*) where, in the non-modular from, there is no need to keep a record since every time the cell needs a value from its neighbor (y), it access it directly. This will make the internal cell transition function defined for state variables of each cell as well as the state variables of its neighboring cell since it has access to all of them according to the coupling relation.

For boundary cells:

$$X^b = \underset{self \in I_i}{\times} X^b_{self,i} = \underset{self \in I_i}{\times} \{sv^k_x \mid (k \in v_{xi}) \wedge (1 \le k \le n)\}$$

$$Y^b = \underset{i \in I_{self}}{\times} Y^b_{i,self} = \underset{i \in I_{self}}{\times} \{sv^k_i \mid (k \in v_{iy}) \wedge (1 \le k \le n)\}$$

Where v_{xi} is the set of state variables needed to be received by boundary cells through external input ports and v_{iy} is the set of state variables needed to be sent by boundary cells through external output ports. The state variables sv^k_x are used as storage for the external values that are received by the cell space through input ports and they are only accessed by the boundary cells. On the other hand, the external outputs that are required to be sent out of the cell space are the states variables of the boundary cells.

One more issue in the equivalency to the non-modular form is that the neighboring cells do not always have the last updated values. One reason for that is using the quantized DEVS in which the cell does not inform the neighboring cells with its last modification if the difference is not above a specific quantum. The above equivalency analysis is correct if we set the quantum to zero. However, if it is not zero, each cell should keep two copies of state variables (e.g. now and new). Whenever a cell needs to access a value in its neighboring cell, it will access the (now) value which represents the last value that crossed the quantum level (i.e. was sent to the cell through ports in the modular form). The (new) value is the last updated value of the cell which is kept different from (now) till it crossed the quantum level and the change will be committed to (now) to be available for other cells to access. Now, we can redefine the cell transition functions as follows:

For each specific cell i :

δ^{*}_{int}: $S_{i\text{-}now} \rightarrow S_{i\text{-}new}$ *where* $s_i \in S_i$ *given* $s_i=(sv^{1}_{i}, sv^{2}_{i}, sv^{3}_{i}, \ldots sv^{n}_{i})$

δ^{*}_{ext}: $\underset{j \in I_i}{\times} S_{j-now} \times Q_i \rightarrow Q_i$

δ^{*}_{con}: $\underset{j \in I_i}{\times} S_{j-now} \times Q_i \rightarrow Q_i$

λ^{*}: $S_{i\text{-}new} \rightarrow S_{i\text{-}now}$

The output function for each cell just updates the state variables of the current cell in case it exceeds the quantum level and there is no need to generate any outputs if the cell is not a boundary cell. This task is actually done in the internal transition function δ^{*}_{int} and we can select not to duplicate the task. Another reason is to keep the new specification consistent with the DEVS specification where the output function is used to send messages only and does not initiate state or variable changes in the model. Therefore, δ^{*}_{int} will commit the changes in variables and add the neighboring cells to the scan group *EXT*.

```
λ(s) :        for all $\{ i \mid i \in IMM \wedge i \in I_{self} \}$
                  Apply $\lambda^{*}$ to $Cell_i$  $[\, Y = Y \cup Z_{i,self}(\{ sv_i^k \mid (k \in viy) \wedge (1 \le k \le n)\}) \,]$
δint(s) :     for all $i \in IMM$
                  if $s_{i\text{-}new} - s_{i\text{-}now} > quantum$
```

```
                for all $\{ i | \; i \in I_j \wedge j \neq self \}$
                        If $j \in IMM$  $CONF = CONF \cup \{ j\}$
                        Else $EXT = EXT \cup \{ j\}$
        $s_{i\text{-}now} = s_{i\text{-}new}$
$INT = IMM - CONF$
Update EVENTS:          time=time-ta(s) for all ( time, i) $\in$ EVENTS
                        delete any ev=( 0 , i ) where ev $\in$ EVENTS
for  all $i \in INT$:        apply $\delta^{*}_{\text{int}}$ to $Cell_i$
                        schedule (next ta, i)
for  all $i \in EXT$:        apply $\delta^{*}_{ext}$ to $Cell_i$
                        schedule (next ta, i)
for  all $i \in CONF$:       apply $\delta^{*}_{con}$ to $Cell_i$
                        schedule (next ta, i)
clear all cell groups INT=EXT=CONF={}
if (EVENTS $\neq$ {})
        extract IMM  from EVENTS
        ta(s) = minimum {time |( time, i) $\in$ EVENTS }
else
        clear IMM={}
        ta(s) = $\infty$
```

```
$\delta_{con}(s,x^b)$ :    for all $i \in IMM$
        if $s_{i\text{-}new} - s_{i\text{-}now} > quantum$
            for all $\{ i |\ i \in I_j \wedge j \neq self \}$
                If $j \in IMM$  $CONF=CONF \cup \{ j\}$
                Else $EXT=EXT \cup \{ j\}$
        $s_{i\text{-}now} = s_{i\text{-}new}$
    $INT= IMM - CONF$
    $CONF'=CONF \cup \{\{INT\} \cap \{ j \mid self \in I_j \wedge Z_{self,i}(X^b) \neq \emptyset\}\}$
    $INT'=INT - \{\{INT\} \cap \{ j \mid self \in I_j \wedge Z_{self,i}(X^b) \neq \emptyset\}\}$
    $EXT'=EXT \cup \{\{ j \mid self \in I_j \wedge X^b_j \neq \emptyset\} - CONF'\}$
    Update EVENTS:    time=time-ta(s) for all ( time, i) $\in$ EVENTS
                      delete any ev=( 0 , i ) where ev $\in$ EVENTS
    for all $i \in INT'$ :    apply $\delta^*_{\text{int}}$ to $Cell_i$
                      schedule (next ta, i)
    for all $i \in EXT'$ :    $S_{self\text{-}now}= S_{self\text{-}now} \cup Z_{self,i}(X^b)$
                      apply $\delta^*_{ext}$ to $Cell_i$
                      schedule (next ta, i)
    for all $i \in CONF'$ :   $S_{self\text{-}now} = S_{self\text{-}now} \cup Z_{self,i}(X^b)$
                      apply $\delta^*_{con}$ to $Cell_i$
                      schedule (next ta, i)
    clear cell groups  INT=EXT=CONF=CONF'=INT'=EXT'={}
```

```
                if (EVENTS ≠ {})
                        extract IMM from EVENTS
                        $ta(s) = minimum \{time \mid (time, i) \in EVENTS\}$
                else
                        clear IMM={}
                        $ta(s) = \infty$

$\delta_{ext}(s,e,x^b)$ :    Update EVENTS:        $time=time-e$ for all $(time, i) \in EVENTS$
                for all $i \in \{ j \mid self \in I_j \wedge Z_{self,i}(X^b \neq \emptyset)\}$
                                $S_{self\text{-}now} = S_{self\text{-}now} \cup Z_{self,i}(X^b)$
                                apply $\delta^*_{ext}$ to $Cell_i$
                                schedule (next ta, i)
                if (EVENTS ≠ {})
                        extract IMM from EVENTS
                        $ta(s) = minimum \{time \mid (time, i) \in EVENTS\}$
                else
                        clear IMM={}
                        $ta(s) = \infty$
```

3.3.3 Final Non-Modular Decomposed Format

The last detailed specification above shows that the cell internal output function tasks were encapsulated under the cell space output function and there is no need to define λ^* as an independent function. Similarly, δ^*_{ext} or δ^*_{con} are not defined for non-modular cells as shown in Figure 3.1.

Assumption-2:

The modular cells use δ^*_{ext} and δ^*_{con} to update the values of their neighboring states and then apply δ^*_{int} to make calculations and transitions according to the updated values. This means that δ^*_{ext} and δ^*_{con} are designed not to make calculations or processing, but force the cell to do an internal transition which considers the new updates.

$$\delta^*_{ext}(s_i,e_i,x^b_i)=(\{\text{“re-calculate”},sv_i^1, sv_i^2,\dots, sv_i^n\},0, \underset{j\in I_i}{\times}\{sv_j^k \mid (k \in vji) \wedge (1 \le k \le n)\}=x^b_i)$$

$$ta(\text{“re-calculate”})=0;$$

$$\delta^*_{con}(s_i,x^b_i)= \delta^*_{int}(\delta^*_{ext}(s_i,ta(s_i),x^b_i))$$

In the non-modular form, each cell will update its own values using the internal transition function δ^*_{int} and then, their neighboring cells will be added to the scanning list which in turn schedules an external event for the neighboring cells. This is exactly what the cell output function λ^* and the cell external transition function δ^*_{ext} do given that assumption-2 is satisfied. Therefore, the equivalency to the modular form is satisfied and there is no need to have the functions λ^*, δ^*_{ext}, and δ^*_{con} explicitly in the resultant model since their tasks are already implied in the new framework. As a result, the model can be simplified for ease of user specification in such a way that each cell has only an internal transition function that is applied every time a cell becomes active. In addition, the cell groups can be merged as we do not distinguish between the different groups. However, in addition to the *IMM* group, we still need the *EXT* group in order to update the neighboring values of the imminent cells. Note that from now on, we will refer to the cell internal transition function δ^*_{int} as the cell's local transition function Δ^* ($\Delta^*=\delta^*_{int}$).

For each specific cell i :

$$\Delta^*: \bigtimes_{j\in I_i} S_{j-now} \times S_{i-now} \rightarrow S_{i-new} \quad \text{where } s_i \in S_i \text{ given } s_i=(sv^1_i, sv^2_i, sv^3_i, \ldots sv^n_i)$$

```
$\lambda(s)$ :      for all $\{ i \mid i \in IMM \wedge i \in I_{self} \}$
                        $Y = Y \cup Z_{i,self}(\{sv_i^k \mid (k \in viy) \wedge (1 \le k \le n)\})$

$\delta_{int}(s)$ :     Update EVENTS:    $time = time - ta(s)$ for all $(time, i) \in EVENTS$
                                      delete any $ev = (0, i)$ where $ev \in EVENTS$
                    for all $i \in IMM$
                        if $s_{i-new} - s_{i-now} > quantum$
                            for all $\{ i \mid i \in I_j \wedge j \neq self \}$    $EXT = EXT \cup \{j\}$
                        $s_{i-now} = s_{i-new}$
                    for all $i \in \{IMM \cup EXT\}$ :
                                      apply $\Delta^*$ to $Cell_i$
                                      schedule (next ta, i)
                    clear cell group $EXT = \{\}$
```

```
                 if (EVENTS $\neq$ {})
                         extract IMM from EVENTS
                         $ta(s)$ = minimum {time |( time, i) $\in$ EVENTS }
                 else
                         clear IMM={}
                         $ta(s) = \infty$
$\delta_{con}(s,x^b)$ :   Update EVENTS:      time=time-ta(s) for all ( time, i) $\in$ EVENTS
                                         delete any ev=( 0 , i ) where ev $\in$ EVENTS
                 for all i $\in$ IMM
                         if $s_{i\text{-}new} - s_{i\text{-}now} >$ quantum
                                 for all { i| i $\in I_j \wedge j \neq self$ }    EXT=EXT $\cup$ { j}
                         $s_{i\text{-}now} = s_{i\text{-}new}$
                 EXT=EXT $\cup$ { j | self $\in I_j \wedge X^b{}_j \neq \emptyset$}
                 for all i $\in$ { i | self $\in I_i \wedge X^b{}_i \neq \emptyset$} :    $S_{self\text{-}now} = S_{self\text{-}now} \cup Z_{self,i}(X^b)$
                 for all i $\in$ {IMM $\cup$ EXT} :
                                         apply $\Delta^*$ to $Cell_i$
                                         schedule (next ta, i)
                 clear cell group EXT={}
                 if (EVENTS $\neq$ {})
                         extract IMM from EVENTS
                         $ta(s)$ = minimum {time |( time, i) $\in$ EVENTS }
```

```
        else
                clear IMM={}
                $ta(s) = \infty$
$\delta_{ext}(s,e,x^b)$ :  Update EVENTS:  $time=time-e$ for all $(time, i) \in EVENTS$
        for all $i \in \{ j \mid self \in I_j \wedge Z_{self,i}(X^b \neq \emptyset)\}$
                        $S_{self-now} = S_{self-now} \cup Z_{self,i}(X^b)$
                        apply $\Delta^*$ to $Cell_i$
                        schedule (next ta, i)
        if (EVENTS $\neq \{\}$)
                extract IMM from EVENTS
                $ta(s) = minimum \{time \mid (time, i) \in EVENTS\}$
        else
                clear IMM={}
        $ta(s) = \infty$
```

This last specification is now decomposed fully into the non-modular form. An atomic cell space now consists of identical cells having the same transition function at the cell space level but covering different sets of data and state variables. The cell space was converted into an atomic non-modular P-DEVS as shown above given that assumption-1 and assumption-2 are satisfied. The resultant model can be further simplified and put in the following format:

```
$\lambda(s)$ :        for all $\{ i \mid i \in IMM \wedge i \in I_{self} \}$
```

```
        $Y = Y \cup Z_{i,self}(\{sv_i^k \mid (k \in viy) \wedge (1 \le k \le n)\})$

$\delta_{int}(s)$ :    Update EVENTS:    $time = time - ta(s)$ for all $(time, i) \in EVENTS$
                                  delete any $ev = (0, i)$ where $ev \in EVENTS$
    for all $i \in IMM$
        if $s_{i\text{-}new} - s_{i\text{-}now} > quantum$
            for all $\{ i \mid i \in I_j \wedge j \neq self \}$    $EXT = EXT \cup \{j\}$
        $s_{i\text{-}now} = s_{i\text{-}new}$
    for all $i \in \{IMM \cup EXT\}$ :
                                  apply $\Delta^*$ to $Cell_i$
                                  schedule (next ta, i)
    clear cell group $EXT = \{\}$
    if ($EVENTS \neq \{\}$)
        extract IMM from EVENTS
        $ta(s) = minimum \{time \mid (time, i) \in EVENTS\}$
    else
        clear $IMM = \{\}$
        $ta(s) = \infty$
$\delta_{con}(s,x^b)$ :    conf = true
    $\delta_{ext}(s,e,x^b)$
    $\delta_{int}(s)$
    conf = false
```

```
$\delta_{ext}(s,e,x^b)$ :   if (!conf)  Update EVENTS:   time=time-e  for all ( time, i) $\in$ EVENTS
                            $EXT = \{ j \mid self \in I_j \wedge Z_{self,i}(X^b \neq \emptyset)\}$
                            for  all i $\in$ EXT
                                        $S_{self\text{-}now} = S_{self\text{-}now} \cup Z_{self,i}(X^b)$
                            if (!conf)  IMM = EXT
                                        ta(s)=0          // fire $\delta_{int}(s)$ to make the state transitions
```

3.4 A Proposition to Show the Generality of the Approach

The generality of the approach that follows the closure under coupling property in parallel DEVS was ensured in all steps in the procedure with no constraints except the two assumptions introduced in subsections 3.3.2 and 3.3.3. The final format reached assumes that the models satisfy those assumptions. This section shows and proves the generality of the approach that spans all cellular DEVS models: even the ones that do not agree with the assumptions. This section shows that any modular cell in cellular DEVS model can be modified to satisfy the two assumptions as follows.

A modular cell (*i*), that does not satisfy assumption-1, sends an output value *y* that is not among the cell's state variable values s_i where $s_i \in S_i$. That value will either be a fixed value for each cell, fixed parameter, or cell's calculated value where $y \notin s_i$. It was shown in section 3.3.2 that $s_i=(sv^1{}_i,\ sv^2{}_i,\ sv^3{}_i,\ \ldots\ sv^n{}_i)$ where *n* is the number of state variables in the cell model. To satisfy assumption-1, the modular cell (*i*) can be updated to extend the state variables representation in order to include one more state variable that account for the values *y*. As a result $s_i=(sv^1{}_i,\ sv^2{}_i,\ \ldots\ sv^{n+1}{}_i)$ where $sv^{n+1}{}_i$ is a newly added state variable to the model that stores the value of y which can be then sent through external ports according to assumption-1.

A modular cell (i), that does not satisfy assumption-2, does some computations and state transitions in δ_{ext}. Operations in this function can be separated into two phases in any DEVS cell model. The first one accepts the external values on ports and updates the state variables accordingly in zero time. The other phase will include the calculations and state transitions that are based on the updated values.

$\delta^{*}_{ext}(s_i,e_i,x^b{}_i) = s_{i\text{-}new} \rightarrow \delta^{*}_{ext}(s_i,e_i,x^b{}_i) = \delta^{*}_{ext}(s_{i\text{-}temp},e_i) = s_{i\text{-}new}$

First phase:

$(\underset{j\in I_i}{\times}\{sv_j^k \mid (k \in vji) \wedge (1 \le k \le n)\} = x^b{}_i) + s_i \rightarrow s_{i\text{-}temp}$

$s_{i\text{-}temp} = \{$"re-caculate", $sv_{i\text{-}temp}{}^1, \ldots, sv_{i\text{-}temp}{}^n\}$

Second phase:

$\delta^{*}_{ext}(s_{i\text{-}temp},e_i) = s_{i\text{-}new} = \{sv_{i\text{-}new}{}^0, sv_{i\text{-}new}{}^1, \ldots, sv_{i\text{-}new}{}^n\}$.

To satisfy assumption-2, the second phase should be moved into the internal transition function that should now deal with a temporary state "re-calculate". The first phase is left in the external function to update state variables based on the received messages. Therefore, the external transition function can be defined in the following format:

$\delta^{*}_{ext}(s_i,e_i,x^b{}_i) = (\{\text{"re-calculate"}, sv_i^1, sv_i^2, \ldots, sv_i^n\}, 0, \underset{j\in I_i}{\times}\{sv_j^k \mid (k \in vji) \wedge (1 \le k \le n)\} = x^b{}_i)$

which account for the new updates and sets the time advance to be zero in order to fire the internal transition function in the next step that will deal with the temporary state "re-calculate" on the updated values and hence satisfying assumption-2. Similar approach can be done in the confluent function.

3.5 Fast Cellular DEVS Specification

Based on the full decomposition process, we can now introduce a specification (i.e. formalism layer) to represent atomic DEVS cell space that will run faster than the conventional implementations that are based on representing each cell as an atomic model. Therefore, a cell space can be formed in the following P-DEVS atomic structure:

Atomic $CellSpace = \langle X, S, Y, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events \rangle$.

Where,

D is the set of cell ID's encapsulated in this atomic model, |D| = number of cells in the model

B is the set of the boundary cells ID's

$Cell_{id} = \langle n, S^* \rangle$ for all $id \in D$ and n is the set of neighboring cells and S^* the state variables set (in non-modular form) where $s^*=(sv_1, sv_2, sv_3, \ldots..) \mid sv_n$ is the value of the nth state variable of the cell for a given $s \in S^*$.

X is a set of input values defined only for boundary Cells $id \in B$ (set of ports/values in coupled structures).

Y is a set of output values.

S is a set of general states of the atomic model

The total state set $Q = \{(s, e, \{\underset{id \in D}{\times} Q^*_{id}\}) \mid s \in S, 0 \le e \le ta(s)\}$

$Q^*_{id} = \{(s^*_{id}, e_{id}) \mid s^* \in S^*, 0 \leq e_{id} \leq ta(s^*)_{id}\}$

δ_{int}: $Q \to Q$ is the *internal transition* function.

δ_{ext}: $Q \times X^b{}_B \to Q$ is the *external transition* function,

δ_{con}: $Q \times X^b{}_B \to Q$ is the *confluent transition* function,

λ: $Q \to Y^b$ is the output function, only for cells with $id \in B$

Events: is the next events list where $ta=min\{time|(time,id) \in Events\}$

The four functions are executed iteratively and efficiently as explained at the end of section 3.3.3.

For multi cell spaces, we can couple more than one atomic cellular space in one coupled model which will be in the form of parallel coupled DEVS:

$CM = \langle X, Y, D, \{CellSpace_i\}, \{I_i\}, \{Z_{i,j}\} \rangle$.

In addition, the specification can be extended to d-dimensional cell space as follows:

$CellSpace^d = \langle X, S, Y, \{N_i\}, d, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events \rangle$.

Where, d is the dimension of the cell space and N_i is the number of cells in the ith dimension given $1 \leq i \leq d$. Then, *id* will be a set of IDs for cell in each dimension $\{id_i\}$. In array implementation of the cell space atomic mode, this update will result in d-dimensional state arrays for the state variables. However, in that case, the cell's id will be the set of array indexes in each dimension.

Example:

In the case of 2-D cell space, id=(i,j) where i is the cell id in the first dimension and j in the other dimension. n={(i*, j*)| i* neighbor to i $\vee$ j* neighbor to j} is set of 2D neighbors

$Cell_{id} = Cell_{(i,j)} = \langle n, S^* \rangle$ for all $id \in D$

B is the set of boundary cells IDs = $\{(i,j) \mid i=1 \vee i=N_1 \vee j=1 \vee j=N_2\}$

N_1 is number of cells in the first dimension, N_2 in the other dimension where the cell space size will be $N_1 \times N_2 = |D|$.

$B \subseteq D$ and $|B| = 2N_1 + 2N_2 - 4$

Lemma 1

CellSpace is a parallel DEVS (P-DEVS) atomic Model.

Proof:

The formalism *CellSpace* was generated from the closure under coupling property. The step by step procedure ensured that the new representations keep the general P-DEVS structure and hence equivalency.

Given that

$$CellSpace = \langle X, S, Y, \{N_i\}, d, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events \rangle$$

and

$$\text{P-DEVS} = \langle X, S, Y, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

By analyzing the definitions of both models, it can be seen that

$CellSpace.X \equiv \text{P-DEVS}.X$

$CellSpace.Y \equiv \text{P-DEVS}.Y$

By definition, S is the set of states which includes all state variables as well. Therefore, P-DEVS.S={Phase×{P-DEVS.S^*}}, where {S^*} is the total set of all internal state variables in the model and phase is the representing state of the model (string). Equivalently, *CellSpace*.S={Phase×{ $\underset{id \in D}{\times}$ $\text{Cell}_{\text{id}}.S^*$}}, where the cell's S^* is the set of state variables inside that cell. Since S can be equivalently set for both models, the internal transition function as well as the output functions are defined for the same S where δ_{int}: S→S and λ: S→Y (Moore type). In addition, δ_{ext} and δ_{con} are applied to the total state set Q and the time advance of *CellSpace* is defined as the minimum time in the Events list. Therefore, both models are equivalent with more details and parameters in *CellSpace* which can be implied in the internal behavior of any P-DEVS atomic model.

Lemma 2

A DTSS model can be represented by *CellSpace*.

Discrete Time System Specification (DTSS)

A Discrete Time System Specification is a structure:

DTSS = $\langle X, Y, Q, \delta, \lambda, c \rangle$, where X is the set of inputs, Y is the set of outputs, Q is the set of states, $\delta : Q \times X \rightarrow Q$ is the state transition function, $\lambda : Q \rightarrow Y$ is the output function (Moore-type), and c is a constant employed for the specification of the time base $c \bullet \mathfrak{I}$.

Proof:

Since *CellSpace* is an atomic P-DEVS model, all what we need is to prove that a DTSS model can be represented by P-DEVS. Then, by induction the new framework can represent the DTSS.

P-DEVS = $\langle X, S, Y, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$ can be put in the general I/O system structure $\langle T', X', \Omega', Y', Q', \Delta', \Lambda' \rangle$ where the time base *T* is the real numbers R, input set is $X' = X^b \cup \{\emptyset\}$ (i.e., input set of the dynamic system is the input set of the DEVS together with the nonevents set specified by $\emptyset \notin X^b$), output set is $Y^{\emptyset} = Y \cup \{\emptyset\}$, state set $Q = \{(s,e) \mid s \in S,\ 0 \leq e \leq ta(s)\}$ is the total state set, the set Ω of admissible input segments is the set of all DEVS segments over X^b and *T, t*he state trajectories are piecewise constant segments over S and T, and the output trajectories are DEVS segments over Y^b and T. For equivalency of the two structures: $X \equiv X'$, $Y \equiv Y'$, $ta(s) \in T'$, $Q' = (S \times T')$, $\Delta' = \delta(s,e,x^b)$ which is the local transition function of the atomic DEVS model as defined in 3.3.3, and finally, the output function is defined as follows:

$$\Lambda' = \begin{cases} \lambda(s) & \textit{if} \quad e = ta(s) \\ \Phi & \textit{otherwise} \end{cases} \quad \textit{... considering Moore type}$$

Similarly, DTSS models can be put in the I/O system structure $\langle T', X', \Omega', Y', Q', \Delta', \Lambda' \rangle$ where the time base T is the set $c \bullet \Im$, the set Ω of admissible input segments is the set of all segments over *X* and *T*, and the equivalency in both structures includes: X, Y and Q.

Now, P-DEVS is capable of presenting DTSS models by first setting the time base to be an integer discrete subset of its general real time base. This will make the DEVS model make transitions and/or generate outputs in discrete steps of time c. This means that all ta(s) as well as the time to receive inputs are either 0 or c. Therefore, the P-DEVS local transition function can be put in the following form:

$$\Lambda' = \delta(s,e,x^b) = \begin{cases} \delta_{ext}(s,e,x^b) & e = 0 \wedge x^b \neq \Phi \\ \delta_{con}(s,x^b) & e = ta(s) = \{0 \vee c\} \wedge x^b \neq \Phi \\ \delta_{\text{int}}(s) & e = ta(s) = \{0 \vee c\} \wedge x^b = \Phi \\ (s,e) & otherwise \end{cases}$$

Which will perform equivalently to the transition function δ of the DTSS over the total set of states that will contain (s,0) or (s,c) and generate an output using λ(s) that is equivalent to the Moore type DTSS λ.

Lemma 3

A cellular model represented by a coupled DTSS formalism can also be represented by an atomic *CellSpace*.

Discrete Time Coupled Models

A discrete time specified network (DTSN) is a coupled system

$N = \langle X, Y, D, \{M_d\}, \{I_d\}, \{Z_d\}, h_N \rangle$

where

X is the set of input values (set of ports/values in coupled structures).

Y is the set of output values (set of ports/values in coupled structures).

D is the set of components.

for each d in D: M_d is a DTSS or FNSS basic component.

for each d in $D \cup \{\text{self}\}$: I_d is the influencees of d, d is not in I_d (no delay-less feedback loop).

self is the coupled model itself N which allow external inputs and outputs.

for each j in I_d : Z_d is the d to j output translation function.

h_N is a constant time advance employed for the specification of the time base $h_N \bullet \mathfrak{I}$ which should be identical to the time base of all of the internal components.

Proof:

A cellular model can be represented in the DTSN formalism by putting it in the following from

$CellSpce_{DTSN} = \langle X, Y, D, \{Cell_{id}\}, \{I_{id}\}, \{Z_{id}\}, h_N \rangle$ where each cell with id $\in$ D is a DTSS atomic model that is $\mathrm{Cell}_{\mathrm{id}} = \langle X, Y, Q, \delta, \lambda, c \rangle$ with the time step $c=h_N$. The DTSS formalism was proved to be closed under coupling in [6]. That is a DTSN model can be presented in an equivalent DTSS atomic model. That means, a cellular space model that is represented by DTSN can be put in the atomic form *CellSpace_{DTSS}*$=\langle X, Y, Q, \delta, \lambda, c\rangle$ and following Lemma 2, this resultant model can be presented in *CellSpace*.

We can also prove this by following the same decomposition procedure presented in the previous sections but this time for converting a cellular coupled model in DTSN into DTSS atomic model with all details. Then as a special case for the DTSS, the scheduling algorithm and processing of cells is done in discrete time step of duration c.

Example: 2-D Game of Life

$GOL = \langle X, Y, D, \{Cell_{id}\}, \{I_{id}\}, \{Z_{id}\}, h \rangle$ with time step h=1, X={}, Y={}, and $id=(i,j)$ where i is the id in the first dimension and j in the other. Each cell is a DTSS model that has $\langle X_{(i,j)}, Y_{(i,j)}, Q_{(i,j)}, \delta, \lambda, h\rangle$, where δ, λ, and h are identical for all cells.

$Q_{(i,j)}=\{0,1\}$, $Y_{(i,j)}=\{0,1\}$, $X_{(i,j)}=\{0,1\}$

Using Moore output type: $y = \lambda(q) = q$

$$\delta(q_{(i,j)}, X_{(i,j)}) \quad = \begin{cases} 1 & if\left(\left(\sum X_{(i,j)} = 3\right) \vee \left(\left(q_{(i,j)} = 1\right) \wedge \sum X_{(i,j)} = 2\right)\right) \\ 0 & otherwise \end{cases}$$

$I_{(i,j)}=\{ (i-1,j-1), (i-1,j), (i-1,j+1), (i,j-1), (i,j+1), (i+1,j-1), (i+1,j), (i+1,j+1) \}$

$Z_{(i,j)}: Y_{(i,j)} \rightarrow X_{(i',j')} \; for \; (i',j') \in I_{(i,j)}$

At each time step h, a cell will generate a transition and send its current state q to all of its eight neighboring cells. Therefore, each cell will receive 8 binary values at X

that represent the binary states of its 8 neighbors which will be summed up in the transition function.

Representing GOL in *CellSpace*

GOL^1 = $\langle X, S, Y, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events\rangle$ with $Cell_{id} = \langle n, S^* \rangle$

Similarly, *X*={}, *Y*={}, and for each cell: *id=(i,j)*, S^*={0,1}, and *n={ (i-1,j-1), (i-1,j) , (i-1,j+1) , (i,j-1), (i,j+1), (i+1,j-1), (i+1,j) , (i+1,j+1) }. S={"active"}*

B={} since there is no need for any cell to send or receive at boundaries. The basic functions of this atomic P-DEVS model δ_{int}: active → active, δ_{ext}, δ_{con}, and λ fixed iterative functions those where defined in section 3.3.3 with a specific cell's local transition function

$$\Delta^*(i,j) = \begin{cases} 1 & if\left(\left(\sum_{id\in Cell_{(i,j)}.n} Cell_{id}.s^* = 3\right) \vee \left(\left(Cell_{(i,j)}.s^* = 1\right) \wedge \sum_{id\in Cell_{(i,j)}.n} Cell_{id}.s^* = 2\right)\right) \\ 0 & otherwise \end{cases}$$

With all time advances equal to *h* in GOL^1. That means that all times in Events list are equal to *h* and hence *ta(active)=h* in all iterations.

3.6 Solving Differential Equations Using Cell Space Models

A differential equation is any equation that contains derivatives, either ordinary or partial. Numerical solutions of differential equations consider approximating the derivatives of these equations in order to find a particular solution for given initial values. This may include discretizing the equations spatially or/and temporally. In recent years, many researchers dedicated their work to employ modeling and simulation theories in solving differential equations. As a result, different formalisms and methods where developed to precisely represent differential equations as system models and generate their solutions by simulating those models.

Depending on the discretization methods, the modeler can select the proper formalism to present and solve his differential equations. For example, Differential Equation System Specification (DESS) represents equations with the minimum state discretization required for computer representation, where Discrete Time System Specification (DTSS) represents them with spatial and temporal discretization. Quantized DEVS was introduced to solve equations on continuous time space with discrete quantized states. These approximation/discretization methods, of course, introduce some errors compared to their analytical solutions. However, a 100% accurate solution is impossible to achieve in computer generated solutions since they cannot handle continuous state representations (i.e. finite state machine). Therefore, the modeler's task is to select a representation that minimizes the error and speeds up computations (i.e. generates the solution in the fastest time possible).

Representing differential equations in cell space models requires dividing them spatially into cells having identical representative structures where each cell stores the state variables of the area it covers and applies the transition functions to generate the solution for that specific area. Now, the modelers will differ in selecting to discretize these cell's states as well as their temporal domain or not. Discretizing them using a fixed time step will result in a network of DTSS cells, where the variable time step (i.e. time to next state transition) will result in a network of atomic DEVS cells with discrete input/output events. On the other hand, continuous input/output events requires using cells in the form of quantized atomic DEVS which is the closest, but faster, representation to the DESS, which apparently does not discretize temporal space.

Lemma 4

CellSpace is able to represent and solve spatial differential equations numerically.

Proof:

Using numerical methods, any differential equation can be approximated in a difference equation form. This difference equation is spatially divided into smaller units that can be put in a mesh form which in turn can be put in a cell space model format. Then, each cell can be presented in the form of DTSS with a fixed time step and discrete input/output events. The solution of the equation on the specified space is obtained by gathering the solutions of all cells under the whole coupled network of DTSS cell models. Lemma 2 proved that this coupled model can be presented in *CellSpace*. Therefore, it can represent differential equations and solve them. Furthermore, it is capable of representing the differential equations using a variable time step rather than a fixed time step based on the quantization principles that are inherited from being within the DEVS framework.

3.7 Solving Partial Differential Equations Using the New Framework

Since most of the complex natural system's dynamics can be formulated using partial differential equations, in this section, we are going to give examples on how to solve such equations using cellular space models. The two dimensional cell spaces are the most commonly used in representing three dimensional phenomena since the third dimension can be translated into a state variable that is dependent on two spatial dimensions as well as a temporal dimension. For example, in land elevation models, a land height (Z) at any given time (t) is specified by a value (state variable) that is given according to a specific point described by x-y coordinates in the cell space. In this section, we are going to give one example (in 2-D Cell space) considering the new formalism in obtaining solution formulation.

Example (2-D Heat Equation):

Solve the partial differential equation: $\frac{\partial u}{\partial t} = c\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$ using the following

Cell DEVS atomic model: $CellSpace = \langle X, S, Y, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events \rangle$

Solution:

This equation can be represented by a DTSS equivalent DEVS model or quantized state DEVS model.

<u>DTSS like:</u>

Using the difference methods, this equation becomes:

$$\frac{u_{(i,j)}^{n+1} - u_{(i,j)}^{n}}{\Delta t} = c\left(\frac{u_{(i+1,j)}^{n} - 2u_{(i,j)}^{n} + u_{(i-1,j)}^{n}}{(\Delta x)^2} + \frac{u_{(i,j+1)}^{n} - 2u_{(i,j)}^{n} + u_{(i,j-1)}^{n}}{(\Delta y)^2}\right)$$

Equation-Model $=\langle X, S, Y, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events\rangle$, where *X*={}, *Y*={}, *B*={}, *S*={"active"}, *{Cell*$_{id}$*}=Cells*[*M*][*N*] (array of size *M* by *N*), *M* = number of cells in x-axis, *N* = number of cells in y-axis, $M\times N= |D| = (L_x/\Delta x)*(L_y/\Delta y)$, L_x is actual presented space length in x-axis, L_y is actual presented space length in y-axis, for each cell, id is the 2-D arry position *(i,j)*, *n*: 4-neighboring (right/left/up/down) rule, and $S^*=u[i][j]$ is the value of *u* at cell *(i,j)*, δ_{int}: active → active, $ta(active)=\Delta t$ and thus $\{t=\{0, \Delta t\}|\ t\in Events\}$.

At each iteration n, the cell transition function Δ^* will calculate the next heat value *u* (at next Δt) based on its current heat value $u^n(i,j)$ as well as the current heat values at its four neighbors $\{(i-1,j),(i+1,j),(i,j-1),(i,j+1)\}$ according to the following equation:

$$u_{(i,j)}^{n+1} = u_{(i,j)}^{n} + \frac{c}{\Delta t(\Delta x)^2}\left(u_{(i+1,j)}^{n} - 2u_{(i,j)}^{n} + u_{(i-1,j)}^{n}\right) + \frac{c}{\Delta t(\Delta y)^2}\left(u_{(i,j+1)}^{n} - 2u_{(i,j)}^{n} + u_{(i,j-1)}^{n}\right)$$

Assuming that $\Delta x = \Delta y = d$, the equation becomes

$$u_{(i,j)}^{n+1} = \left(1 - \frac{4c}{\Delta t(d)^2}\right)u_{(i,j)}^{n} + \frac{c}{\Delta t(d)^2}\left(u_{(i+1,j)}^{n} + u_{(i-1,j)}^{n} + u_{(i,j+1)}^{n} + u_{(i,j-1)}^{n}\right)$$

Quantized state DEVS:

Each cell in the 2-D space will start with an initial value $u_{(i,j)}(0)$ and then calculate its rate of change:

$$\frac{du_{(i,j)}}{dt} = \frac{c}{d^2}\left(u_{(i+1,j)} - 4u_{(i,j)} + u_{(i-1,j)} + u_{(i,j+1)} + u_{(i,j-1)}\right)$$

Then, the time advance *h* to the next quantum (*q*) level will be calculated according to the following equation which will be used to schedule the next event for that cell.

$$h = \left|\frac{q(d)^2}{c\left(u^n_{(i+1,j)} + u^n_{(i-1,j)} - 4u^n_{(i,j)} + u^n_{(i,j+1)} + u^n_{(i,j-1)}\right)}\right|$$

On event processing (internal or external), a cell should first update its state value $u_{(i,j)}$ according to the previous rate of change as well is its elapsed time since its last event and schedule its next time advance. Then, the cycle goes on till an end point is reached (either stopped by the user or the simulation reached a predefined stopping time). This type is referred to as lazy-DEVS while, on the other hand, aggressive-DEVS updates all cells regardless whether they got events or not. In this work, since we are aiming at high performance, we just consider the lazy-DEVS, which outperforms the aggressive-DEVS. This was taken care of by implementing the scan list, which only considers the imminent cells and updates them while all other cells remain as is.

CHAPTER 4 : A TOOL FOR BUILDING EFFICIENT CELLULAR DEVS MODELS

The objective of this tool is to minimize the coding efforts required by users to develop cellular DEVS models. In DEVSJAVA, the user is responsible for coding all classes, attributes, and functions in order to build a DEVS model that can then be run using the simulation viewer. This actually causes the development process to be longer with a tendency to fall into errors. With the new specifications made in the previous chapter, aiming at faster simulation execution, the coding process will be much more complicated than it used to be. Therefore, developing a new modeling environment becomes a necessity in order to develop efficient cellular space models with minimum development time.

4.1 Ease of User Specifications with GUI support

The new specification presented in section 3.5 was designed to run faster than the conventional cellular space DEVS models. It can be noticed that the functions δ_{int}, δ_{ext}, δ_{con} and λ were well defined (end of section 3.3.3) with no room for further modifications by the modeler except by defining variables, ports, cell's initial data, and the cell transition function, Δ^*, which is then encapsulated to be part of the internal transition function. Therefore, a new specification was introduced for the user to input data that will be used to automatically generate the remaining fixed structure of the full specification. However, at any time, the fixed internal functions, event list handling, and designs can be modified by programmers who seek further speed up if applicable, or seek different interpretation for their own models.

Cell Space = $\left\langle X, S, Y, D, B, \{Cell_{id}\}, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events \right\rangle \rightarrow \left\langle X, Y, D, B, \{Cell_{id}\}, \Delta^{*} \right\rangle$

Where, for each id$\in$D, $Cell_{id} = \left\langle n, S^{*} \right\rangle$.

That resultant specification makes the user just specify the cell's state variables S^{*}, input/output ports sets if applicable, neighboring rules, and the cell's local transition function. The general state variable S for the atomic model does not require special treatment for different kinds of models. It was defined fixed for states that scan the cells, calculate their transitions, schedule next events, and retrieve those next events on their scheduled time. When using arrays to represent cells, the whole atomic cell space will have uniform size arrays of state variables. The cells by the boundaries of these arrays represent the set B and all cells inside those arrays represent the set D with the cell ID's as the array indexes. Cell Space= $\left\langle X, S, \{S^{*}\}, Y, D, B, r, \delta_{\text{int}}, \delta_{ext}, \delta_{con}, \lambda, Events \right\rangle$

As a result, we guaranteed that the development time of cell space atomic models, as well as their run time, becomes much smaller compared to the conventional cellular DEVS specification. Furthermore, this simplicity can be represented in such a way that it supports a graphical user interface (GUI) in the implementation as shown in Figure 4.1. The figure has further details in the user specification, including port to state variable mapping for the automatic code generation. This will make the code decide which values to send or receive at each port and hence this will not be at the user layer of specification.

Following the array implementation, the user specification can be put in the following format:

UserSpecification=$\left\langle X, \{S^*\}, Y, \mu, r, \Delta^* \right\rangle$

Where $\{S^*\}$is the set of the state variables arrays, r is the neighboring rule to be followed (eg. Neumann or Moore neighbors), and μ is set of state variables to ports mapping

$\mu : X' \rightarrow \{S^*\} \rightarrow Y'$ for $X'=\{X \cup \varnothing\}$ and $Y'=\{Y \cup \varnothing\}$

$\varnothing \rightarrow s^* \rightarrow \varnothing$: the values of state variable s^* is not to be sent or received through ports.

$\varnothing \rightarrow s^* \rightarrow y$: the values of s^* can be sent through port y but not expected to receive values. Usually used for statistic gathering ports, not used for neighboring ports.

$x \rightarrow s^* \rightarrow \varnothing$: the variable s^* can get values from port x but does not send values (also not used for neighboring ports)

$x \rightarrow s^* \rightarrow y$: the values of s^* can be sent through port y and expected to receive values from port x

For all the above explanations, what is meant by values are the ones by the boundaries of the arrays only [$(i,j) \in B$]. μ is expected to be a total function over X', Y' and $\{S^*\}$. However, if a state variable is not included in μ, it should be interpreted as $\varnothing \rightarrow s^* \rightarrow \varnothing$ and if a port is not included, the generated code should not send or receive any values over that port.

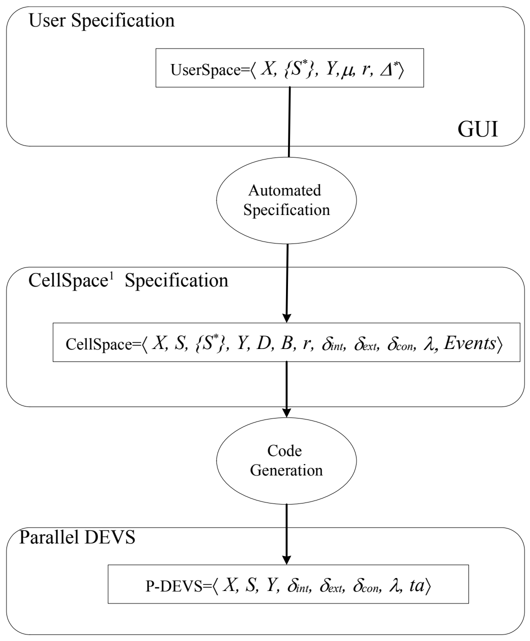

User Specification
UserSpace=$\langle X, \{S^*\}, Y, \mu, r, \Delta^*\rangle$
GUI
Automated Specification
CellSpace[1] Specification
CellSpace=$\langle X, S, \{S^*\}, Y, D, B, r, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, Events\rangle$
Code Generation
Parallel DEVS
P-DEVS=$\langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta\rangle$

Figure 4.1: Specification layers.

The layers presented in Figure 4.1 suggested that the cell space structure is a middle layer which is used to provide information that is sent down to the lower code layer which put them in the Parallel DEVS specification. This design ensured that the new specification is put into the DEVS format and hence any generated model using this design is able to be executed using any parallel DEVS simulator.

4.2 Design Structure of the New Modeling Environment

In Figure 4.1, three layers of specifications were suggested. The lower layer is the parallel-DEVS specification layer which represents the DEVSJAVA modeling and simulation environment. Since it is the targeted running environment for our produced models, all the models to be run should be presented in that specification. Therefore, the new development environment is spanning the two upper specification layers and producing models which must satisfy the third layer of specification. Figure 4.2 shows the design structure of the new cellular DEVS modeling environment which takes model specifications via the Graphical User Interface (GUI) and generates the code that can be run in a DEVSJAVA environment with an efficient cellular DEVS specifications.

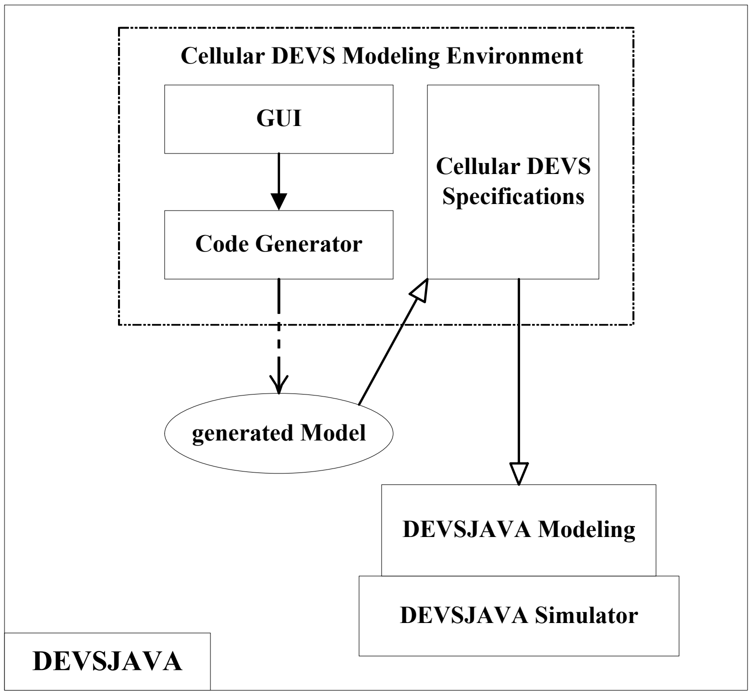


Figure 4.2: Structure of the new cellular DEVS environment.

The main design units in this environment are: GUI, code generator, and the cellular DEVS Specifications unit. Each of these units contains its own classes and has different objectives. Generally speaking, the GUI unit is a DEVS independent unit that gathers information about the model from the user and passes them to the code generator. The code generator unit gathers that information and produces the model’s code guided by the format of the new cellular DEVS specification. The generated code extends the standard classes given by the cellular DEVS specification. Those, in turn, extend the standard modeling classes in the DEVSJAVA modeling layer. The specifications unit hides most of the implementation details and includes huge methods and classes that were hidden from the user to ease the user task in the model development process.

4.3 New Cellular DEVS Specification Unit

This unit contains the basic implementation models for the new specifications previously defined. These are generic models for any standard cell space model to inherit its basic attributes and methods. General Cell space models in DEVS were reviewed guided with two points of view. The first one was to make the generic models implemented in the new efficient specification introduced in the previous chapter. The other one is the view of the user who should be provided with quicker and easier development process following the user specification obtained in section 4.1.

As a result, the implemented classes were designed to minimize the amount of code required by the model developer in specifying the model behavior as generic as possible with some assumptions, constraints and conventions. The main guidelines in building these classes are the homogeneity of the cell space as well as the separation between models, simulator, and experimental frames. The new cell space specifications as well as the conventional one are supported by this environment in order to demonstrate the advantages of the new one as well as to ensure equivalency.

4.3.1 DEVS Cell Space Implementation Models

The user specification layer includes the model information that is required to be entered by the user. The model classes in the specifications unit are DEVS models missing those user specifications. Therefore, the model classes in this unit are the base classes which when provided with user specifications form a complete operational DEVS models. However, they cannot run by themselves without the user specifications since they are missing specific model attributes, cell behavior, cell's initial data, neighboring rule, ports, and port couplings.

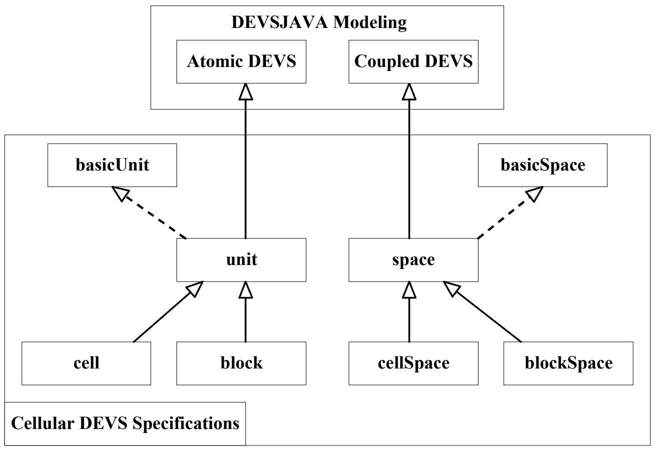


Figure 4.3: Cellular DEVS specification unit.

The main assumption stated in this environment that the developed cell spaces contain identical processing units in two dimensions with the support of modeling one dimension. The support of experimental frames should be external to the space coupled DEVS model. The building block of the cell spaces is the unit which can be a DEVS atomic cell or block, which may contain multiple cells. This will divide the cell spaces into two types: cell space or block space according to what unit type it contains. Figure

4.3 shows the internal structure of the specifications unit which inherits the basic DEVS classes from the DEVSJAVA modeling environment.

4.3.2 Some Implementation Details in Cellular DEVS Models

This section gives an overview on the main rules used to implement the basic model classes and shows the main capabilities they can support.

Neighboring Rules and Neighbors Addressing Convention

The first issue to address is the neighboring rules these models may support as well as neighbors addressing conventions. The standard neighboring rules supported by these models as shown in Figure 4.4 are:

(a) Von Neumann neighboring rule (4-neighbors)

(b) Moore Neighboring rule (8-neighbors)

(c) Hexagonal neighboring rule (6-neighbors)

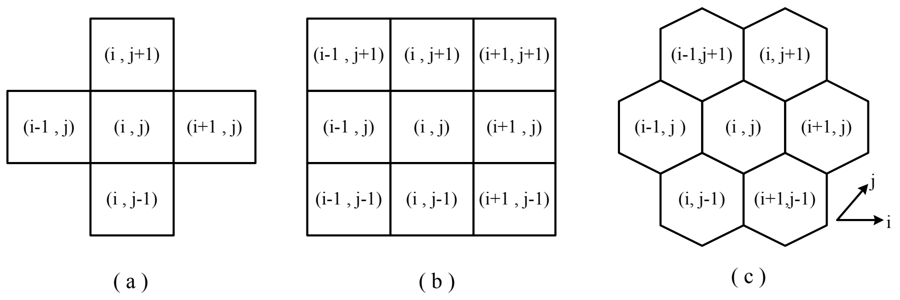


Figure 4.4: Supported neighboring rules.

The addressing convention follows an increasing number scheme, assigned to each of the cell's direction which are 0-3 for Neumann, 0-5 for Hexagonal, and 0-7 for Moore neighboring rules as shown in Figure 4.5. In the implementation models, cells use the method *myNeighbor(int k)* to retrieve the x-y coordinates of its kth neighboring according to addressing scheme mentioned above.

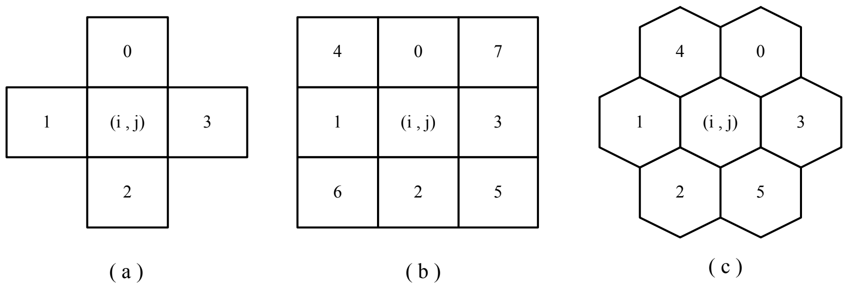


Figure 4.5: Neighbors addressing convention.

Cell Ports Labeling and Couplings

Based on the neighboring rule selected by the user, the cell's structure is then built to handle communication between the neighbors defined by that rule. In the conventional cellular DEVS implementation, cells communicate through ports which can be of a single or multi type. Single port means that the cell sends values to all of it neighbors through one port while in multi type it sends values to each of its neighbors in different ports. The user can provide the development tool with the port name and type. In the cell initialization process, the method *addPorts()* in cell model will be called to add the required ports to the cell structure according to the port names, types and neighboring rule. Table 4.1 shows the result of adding port of name "Port" for different user entries.

Since the ports are added to the cells in a hidden-from-user automated way, coupling the ports is also done in a similar fashion. The mapping factor μ, in user specification, defines the port couplings between cells and which values to send/receive through these ports. Accordingly, the method *doInternalCouplings()*, in cell space model, makes all the necessary couplings between cells. This method is design to account for different types of couplings namely single to single, multi to multi, single to multi, and multi to single port couplings.

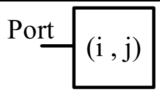
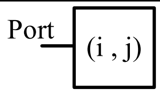
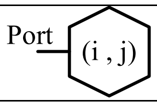
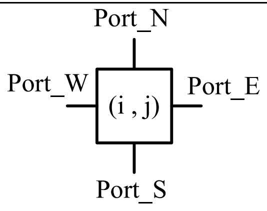
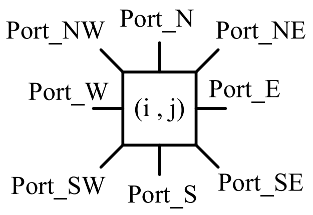
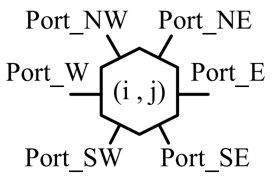
	Neumann	Moore	Hex
Single Port			
Multi Port		 Port_NW Port_N Port_NE Port_W (i , j) Port_E Port_SW Port_S Port_SE	 Port_NW Port_NE Port_W (i , j) Port_E Port_SW Port_SE

Table 4.1: Single vs. multi ports in cell models.

In addition, the mapping specification decides what variable each port gets its values from. The different types of output ports make a distinction between two cell's variable sets: state variables and flow variables. Usually the state variables are connected to a single output port while the flow variables are connected to a multi output port. That is because the state values are fixed for each cell at a given time and there is no need to have a multi port to send a single value to all neighbors while each cell has different flow values for different directions and hence it requires multi output ports to send all flows to neighbors.

Block Ports Labeling

The difference between the cell and the block resides in the fact that a block has multiple numbers of non-atomic decomposed cells encapsulated. Despite the fact that they are all implemented as a DEVS atomic model, the block is equivalent to a DEVS cell space coupled model. Therefore, this big atomic model will have a big list of input/output ports that exchange the values of the boundary cells. The port labeling scheme in block implementation came from the fact that they contain 2-D indexed cell arrays. In addition to the port name, the label should contain which array boundary that specific port is at (i.e. north, south, east, or west) as well as cell index at that boundary (i.e. either x or y coordinate of that cell). Since there are single and multi port types in cell spaces, we should introduce labeling schemes for both types as shown in Figure 4.6 and Figure 4.7. The first scheme is a straight forward example as we explained above since each cell has one port only with an exception for the diagonal cells which are at two boundaries while they have single port. The multi scheme adds a third segment to the labels to indicate the cell neighboring direction since each cell has at least one multi port. Special treatment was done at the corners to include each one of the diagonal ports to each of the major directions in order to avoid redundancy in ports.

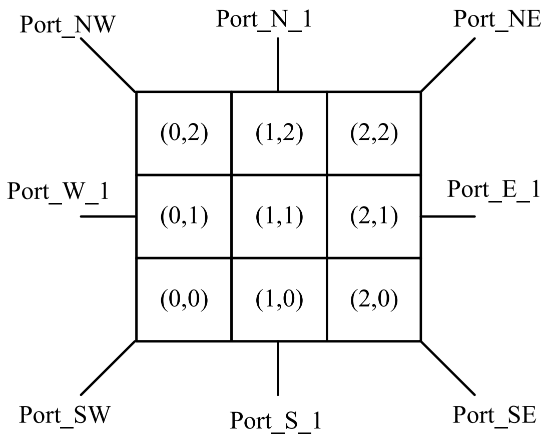


Figure 4.6: Single port labeling scheme.

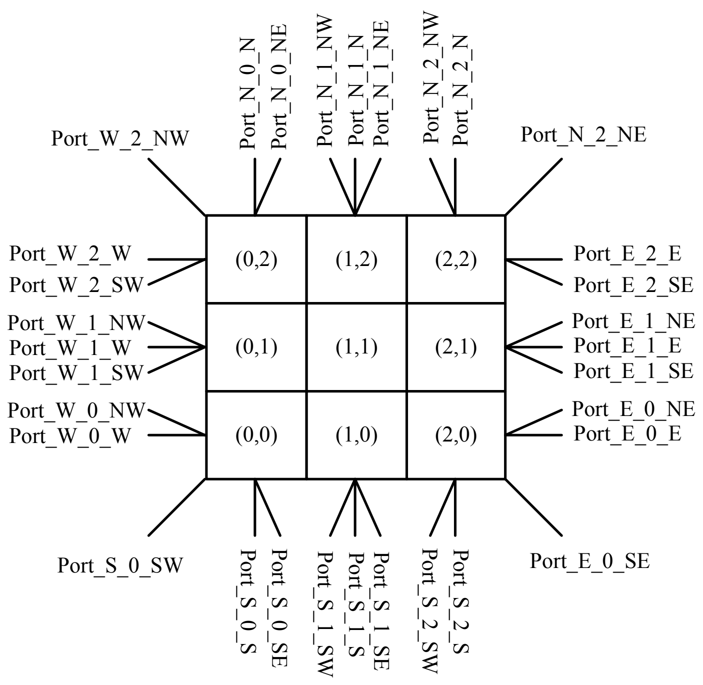


Figure 4.7: Multi port labeling scheme.

Cellular DEVS Space Implementation

The cell space acts like a container for the cells or blocks it covers and provides the means of communicating to other cells in different cell space. In addition, it can provide the external experimental frame with state or statistical data generated by the internal units. It is implemented as a DEVS coupled model that contains atomic DEVS models (i.e. cells or blocks) and it plays a critical role in initiating the internal unit's setup, initialization, and port couplings.

Upon constructing the cell space, it builds all the required internal units using the method *addUnits()*, assigns them IDs, adds the space ports (*addPorts()*) which follows the labeling schemes used in the blocks, makes the internal as well as the boundary port couplings (*doBoundaryCouplings()* and *doInternalCouplings()*),extracts the cell space initial data from an external file, and distributes it to its internal units (*getInitialData()*).

4.4 The GUI

The main class in our environment is the GUI, where the user inputs the cellular model specifications, generates models, and reloads previously generated models to modify. It was implemented using JFrame containing two tabs: the model specifications tab and the cell's local transition function tab. The model specification tab is the main tab, shown in Figure 4.8, which allow the user to write the model name and its package name (i.e. which folder to store the model in), select the space type either cell space or block space, select the neighboring rule to use for his model, select the input data file where the model should extract its initial data from, and fill the ports/variables mapping table. Since the mapping table may have multiple rows, two buttons were introduced for adding and removing rows. The table entries follow the μ segment in the user specification in section 4.1 with additional information regarding the variable type (e.g. integer or double) and ports type (i.e. single or multi). The index column gives the user the ability to specify the order of the variables in the input data file for the model to follow when reading the initial data.

The cell transition function tab was implemented as a text editor that allows the user to write the cell's local transition function Δ^*. Figure 4.9 shows the default text available for the user. The user is required to fill in the first function (*userFunction(i,j)*) which is going to be Δ^*. In case the model needs to use certain quantization schemes or user defined boundary conditions, the user will need to then fill in the other two functions *enforceBoundaryConditions()* and *getQuantum(i,j)*.

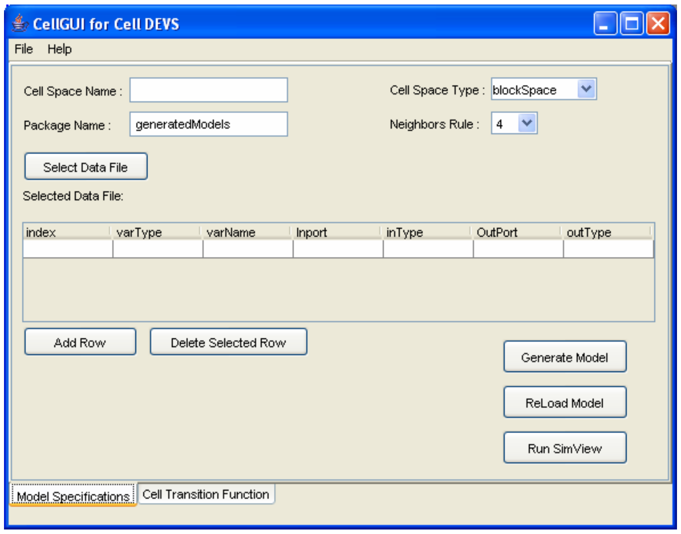
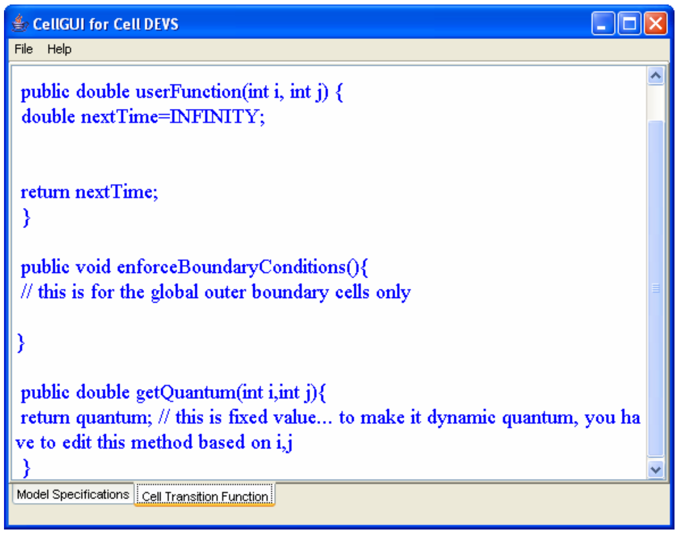


Figure 4.8: The main GUI view.



```
public double userFunction(int i, int j) {
double nextTime=INFINITY;


return nextTime;
}

public void enforceBoundaryConditions(){
// this is for the global outer boundary cells only

}

public double getQuantum(int i,int j){
return quantum; // this is fixed value... to make it dynamic quantum, you ha
ve to edit this method based on i,j
}
```

Figure 4.9: The cell transition function tab.

After entering all model specifications and functions, the user needs to press the generate button which will prompt him to enter the cell space size and the number of blocks for block space type. Then, all model parameters and functions will be sent to the code generator which will generate the code for that model. In addition, the GUI will generate a model property file that stores all user entries which can be used later on to reload them in case the user needs to modify the model. The last button is added to initiate the simulation viewer in order to run the generated models.

4.5 The Code Generator

The main task of the code generator is to write the DEVSJAVA code of the cellular model defined by the user inputs which are received from the GUI. The GUI initiates the abstract code generator class "*codeGenerator*" which may implement, depending on the user selection, the cell space code generator "*cellSpaceCode*" or the block space code generator "*blockSpaceCode*" as shown in Figure 4.10. The abstract class "*codeGenerator*" implements the method *generateCode()* which is called by the GUI to generate the model code. That method calls other methods that are either in the same class or in the other two subclasses. In addition to that method, the abstract class contains the common methods on which the space type has no effect. The required code format of the developed model is essential in defining the code generator tasks and implementations.

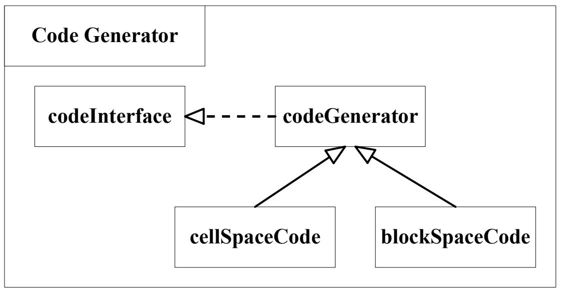


Figure 4.10: Structure of the code generator unit.

4.5.1 The Generated Model Classes

Generic cellular DEVS models usually contain space and cell classes. Any cellular model that is going to be developed should contain two classes inherited from those generic classes. The new approach presented in this work gives the possibility that a model may contain a block class instead of a cell class. Since we refer to cells and blocks as units, we can rephrase the above statement so that any generated model should contain space and unit classes. In this specific environment, the models are designed to read their initial data from an external file and send them to the units as instances of object class. This class is referred to as the model's initial values class which is model specific since it contains the initial values of the model's state variables. Therefore, any generated model should contain code for three classes: model's space class, model's unit class, and model's initial values class. Figure 4.11 illustrates the generated model classes for cell space and block space models.

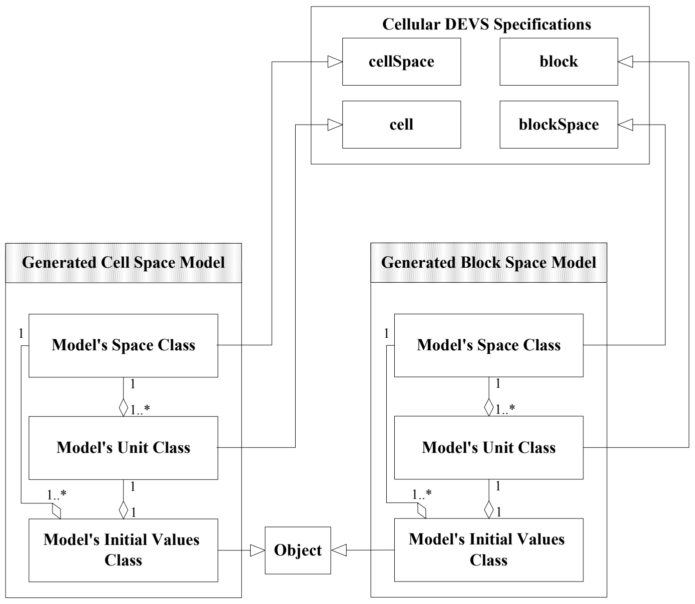

Cellular DEVS Specifications
cellSpace
block
cell
blockSpace
Generated Cell Space Model
Model's Space Class
Model's Unit Class
Model's Initial Values Class
Generated Block Space Model
Model's Space Class
Model's Unit Class
Model's Initial Values Class
Object
1
1..*
1
1

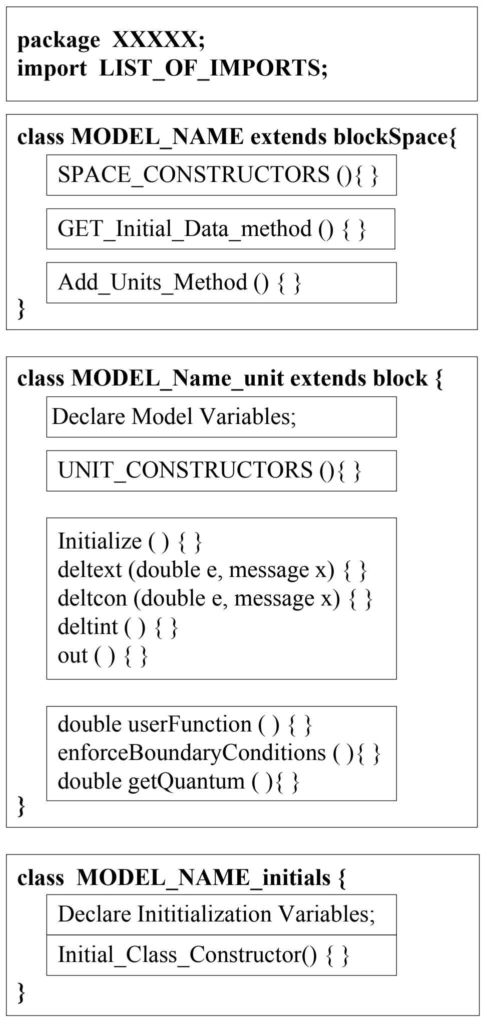
Figure 4.11: Generated model classes and hierarchy.

4.5.2 Some Implementation Issues in the Code Generator

Initially, the code generator is constructed at the GUI with the file name (i.e. where the full source code of the model should be stored) and folder location as specified by the user. The file name is actually generated from the space model name which is also used to name the unit and the initial data classes. Then, the GUI will set all the parameters for the code generator according to the user entries. Therefore, the code generator was designed to have variables that are identical to the ones that should be extracted from the user entries by the GUI.

Figure 4.12 illustrates the content of the source code file that should be produced by the code generator. We can divide it into four segments. The first one is for the package name and the list of imports may be required by the included classes. The list of imports is predefined for the code generator to write in the generated file. The other three segments are for the three required classes as mention in the above subsection.

The model's initial values class code contains the class signature, variable definitions and a constructor that initiates those variables. They are the state variables defined by the user to be read from the input data file. The model's space class code includes the class signature, space constructor, the method that gets the initial data from the external file, and the method that adds the units to the space. The space constructor should include names and values of the input/output ports, neighboring rule, and space size. It should call the following methods in order: *getDataFromFile()*, *addPorts()*, *addUnits(), doInternalCouplings()*, and *doBoundaryCouplings()*.



```
package  XXXXX;
import  LIST_OF_IMPORTS;

class MODEL_NAME extends blockSpace{
    SPACE_CONSTRUCTORS (){ }
    GET_Initial_Data_method () { }
    Add_Units_Method () { }
}

class MODEL_Name_unit extends block {
    Declare Model Variables;
    UNIT_CONSTRUCTORS (){ }
    Initialize ( ) { }
    deltext (double e, message x) { }
    deltcon (double e, message x) { }
    deltint ( ) { }
    out ( ) { }
    double userFunction ( ) { }
    enforceBoundaryConditions ( ){ }
    double getQuantum ( ){ }
}

class  MODEL_NAME_initials {
    Declare Inititialization Variables;
    Initial_Class_Constructor() { }
}
```

Figure 4.12: Illustration of code generation process.

Finally, the code for the unit class should start with the class signature and the rest can be divided into four sub-segments. The first one lists the definition of the model variables and the second one includes the class constructor. The other two sub-segments are the ones which are responsible in stating the model behavior. The third one includes the DEVS atomic functions as well as the initialization procedure. These functions are generated according to the formal specifications in section 3.3.3. As we mentioned in the previous chapter, the internal transition function is the one which is responsible to do all state transitions and call the user functions. Therefore, it should include calls to the user function which are included in the last sub segment. The code generator accepts the user function and prints them, as is, to the generated code.

CHAPTER 5 : DESIGN ISSUES OF THE EVENT LIST

Discrete event simulation proceeds through executing a list of scheduled events that are not simulated yet. Executing an event might result in scheduling other events and the process goes on until the list of events is empty or the simulator reaches a predefined stopping point. In DEVS coupled models, the coordinator is responsible in processing and storing the event list. As a result of converting a coupled model into an atomic model, as is the case in our approach, the event list processing task will be encapsulated inside the new atomic model's functions as described in section 3.3.1. This is achieved by introducing *EVENTS* list which holds scheduled events and then the model processes the cells in the lists extracted from the event list like *IMM* and *EXT*. Consequently, the model will spend large amounts of time just processing the cell list for scheduling future events, extracting current events into *IMM* and *EXT*, and scanning cell lists. Therefore, implementing those cells will have a significant impact on the simulation execution and hence, the targeted speedups.

There are large amounts of related works and debates in the literature to find what are the best processing algorithms and data structures in implementing the event list for discrete event simulations [40-47]. Unfortunately, no single approach is found to work best for all applications. That is because each application has it own event scheduling distribution which will impact the amount and type of operations to execute and hence the execution time of the event list. In addition to the experimental work, [45] and [47] listed a table of the analyzed complexity of certain algorithms that depends on the worst case and average case scenarios independent of application types. The complexity of those algorithms varies from *O(n)* to *O(log n)* which is the best known complexity for operation-wise analysis. There are some amortized bounds which are based on the overall complexity of the event list and not on the execution time of a single operation. However, these bounds cannot be taken as an advantage of some algorithms over others since some of those may take *O(n)* in single operation. Therefore, in this work, two of the common *O(log n)* data structures available, namely binary heap and balanced binary tree [48], were selected for study and analyzed with our own simple straightforward array implementation. Before heading into the design issues and analysis we list the design requirement for the event list.

5.1 Event List Design Requirements

Generally speaking, a DEVS simulation event list should include records of the scheduled events where each record consists of model ID (i.e. place of the event that will occur), time stamp (i.e. when that event should occur), and event type (e.g. external or internal transition). Since the main goal in this work is to develop large scale high performance cell spaces, the list handler should be of an efficient implementation that should process large number of events very quickly to avoid adding latency to the overall cell space atomic model. In opposition to the conventional DEVS simulators [6], our approach is formulated to avoid scanning passive cells which means that those cells should not exist in the event list (i.e. no scheduled records with an infinite time stamp). For all other non-passive cells, a schedule record in the list will contain the cell's x-y coordinates as the model ID and its next time advance as the time stamp. The formulated framework in section 3.3 with its final implementation format at the end of section 3.3.3 suggested that all the cell scheduled events are of internal transition type only. Therefore, the type of the events should be excluded from the scheduled record since by default they are all internal transition events and on expiration of that time advance, the general cell transition function is applied to the scheduled cell. As a result, an event list should include all time advances (ta) of all non-passive cells encapsulated in the atomic model.

The standard operations in event list processing include: adding a new record, and extracting the record with the minimum time stamp which entails finding that minimum time and then deleting that record. Most of the works done in the literature [40-47], in this regard, considered these standard operations only. This is another reason why their findings are inconclusive since there might be additional operations required by different applications. The first additional operation we need in the event list for our approach is the arbitrary removing of cells from the event list. The second one is for advancing the time stamps in all records since the stamps are relative to the local current time of the model. In addition, the method used to extract the least time stamp record might have more operations than just extracting a record. It should search for the minimum time stamp in the list and gather all the cells with that minimum time stamp so that it retrieves it as one cell list *IMM* with the possibility of not removing their records permanently. There is an additional field to the record that might be added in order to emulate the cell's elapsed time. This is required to test if the solution we presented in section 3.3.1 is worth while or we need to include the elapsed time handling in the event list. In that case, another operation will be required to be implemented in the event list which will retrieve the cell history in the list (i.e. how long that cell has been in the list) which is the elapsed time. To summarize, the following operations are required to be included in the event list design except the last one which is optional:

- Add new cell schedule

  Add(ta,i,j) is used to schedule an internal transition event of cell (i,j) after ta time advance. Since an atomic model in DEVS cannot have more than one time advance, redundancy of cell schedule is not allowed here. Whenever a cell is requested to be scheduled, the list should make sure that the cell does not exist in the list. Otherwise, it should delete the previous schedule and store the new one. This might include the *remove(i,j)* operation.

- Remove a cell from the list (arbitrary remove)

  Remove(i,j) is required to delete cell (i,j) from the list. One reason for doing this was stated in the add operation and the other one is when the cell is executed as a neighbor of an imminent cell (i.e. external transition) and generates a new time advance. In this operation, deletion is requested based on the record's value (i,j) rather than the key (time) as the case in most event list implementations.

- Get minimum time in the list

getMin() method searches all the record's keys in the list and returns the minimum key. This minimum time is actually the next time advance for the whole atomic model. Therefore, by the end of any internal transition, the model should call this method to set its next time advance.

- Get the list of imminent cells

getMinList() method will search the event list to return all the cell ID's that have the minimum time in the list. This is also done at the end of the internal transition to get the set of imminent cells *IMM* ready for the next step.

- Remove the imminent cells

removeMin() method should be called after processing the events for all imminent cells. This should include advancing the list time to the minimum time of imminent cells.

- Advance the time of the list

advanceTime(t) should update the event list times with a step of time t. This means that the atomic model advances to the time *t* since the last list time update. This will include subtracting that time *t* from all the time records in the list and adding that *t* to the history records of the cells. Of course, list cannot be advanced to time *t* that is greater than the minimum time available in the list, since the model needs to process records with that minimum time before it advances to *t*.

- Get the elapsed time of scheduled cell

  getHistory(i,j) should return the elapsed time of cell (i,j) since its last state transition. It is actually a historical measure of how long the cell has been in the list.

The above operations should be included in the design of the event list. The system which makes use of it should be capable of calling those methods as defined using the format mentioned above. All other internal algorithms, structures and implementation details in the event list should be independent and hidden from the system using it. This ensures that any event list designed following the above format can perform independently from the application type which allows flexibility in replacing the event list design for different application models. One final issue to mention is that our system was designed to have the imminent cells list *IMM* in a certain format and the method *getMinList()* is supposed to retrieve the list using that format. Therefore, when designing the event list, either the whole design should be consistent with that format or an interface should be introduced to convert to that format by the end of that operation. The first option is not in favor of speedup requirements since it applies constraints on the design and prevents the designer from selecting the fastest algorithms and structures that are using different formats. The better choice is to make the design be as efficient as possible while providing an interface that converts into the specific format at the end.

5.2 Standard Array Implementation

The simplest way of implementing an event list is by using arrays. In array implementation, the records are stored as array entries. Each record will have at least a scheduled time and cell ID. This record might be extended to have the history (i.e. elapsed time) of the cell. In the other alternative, another array will be used to store all history values independent of the main records array. There are two ways to store the event records in the array based event list. The first one is to store them in non-increasing order while the other one does not keep order of records. The non-ordered way makes the add operation of *O(1)* complexity while the other one makes it slower with *O(log n)*, since it is preserving the order, but on the other hand makes the search for the minimum time record *O(1)* compared to *O(n)* in the non-ordered arrays.

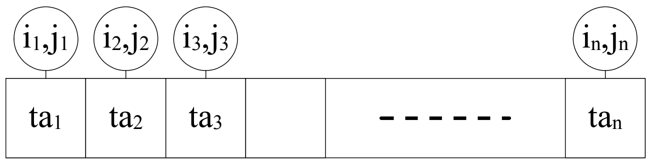


Figure 5.1: Event list in unsorted array.

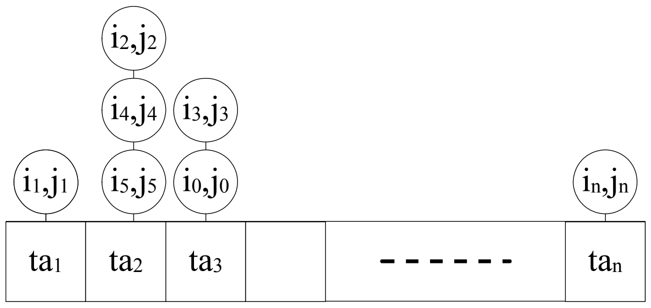


Figure 5.2: Event list in sorted array. ($ta_x > ta_y$ for any x<y)

Figure 5.1 and Figure 5.2 show the two different array implementations of the event list. Since the second implementation keeps the order of records, it is possible to combine the records having the same schedule time as one array entry. If similar implementation is to be done in the unsorted design, upon adding each record a search should be done to find if a record with similar schedule time is already there and in this case it is advantageous to implement the sorted array instead. This idea of combining the records makes getting the list of imminent cells of *O(1)* complexity since it just accesses the last array entry and retrieves the cell list. In addition, it speeds up the process of searching for the least time records since it avoids scanning redundant time values. The reverse order in the sorted array is preferred in order to make the operation of removing the minimum record of *O(1)* complexity. In case of the non-decreasing order, removing the least time record at the beginning of the array requires shifting all other records one step to the right which results in *O(n)* worst case scenario.

The two additional operations added to the event list design which are not usually found in standard event list implementations and analysis are the arbitrary remove and advancing the list time. The advance time operation requires updating all records in the list which results in *O(n)* complexity. However, *n* is the number of array entries which, in the sorted array, is less than or equal the number of recorders or cells (*N*) added to the list. This might make the operation faster compared to the non-ordered array where *n*=*N*. The arbitrary remove operation searches for a specific cell (i,j) in the list and deletes its record. In both array implementation, as discussed so far, searching for that specific record is of *O(n)* complexity since cells are not indexed with their IDs. Improving this search can be done by introducing an indexing scheme to the list which stores a mapping table that maps cell ID's into array location and this will result in *O(1)* search. Then, deleting that record will be of *O(1)* in the case of unsorted array if the record is replaced with the last record in the array. On the other hand, a sorted array should preserve the order and hence all records should be shifted one position to fill the location of the deleted record. As a result, arbitrary operation will have *O(1)* complexity in the case of unsorted array and *O(n)* in the sorted one.

5.3 Binary Heap Implementation

The binary heap is a complete tree structure that is not fully ordered [48, 49]. It is frequently selected and recommended in general purpose implementations as well as event list implementation because of its fair structure, efficiency, and stability [46]. Figure 5.3 illustrates the binary heap structure in which the order of records is kept vertically following the child-parent relation. The key (i.e. time advance) of each node should be greater than or equal to the key of its parent node at all times. As a result, the minimum key value can always be found at the root node.

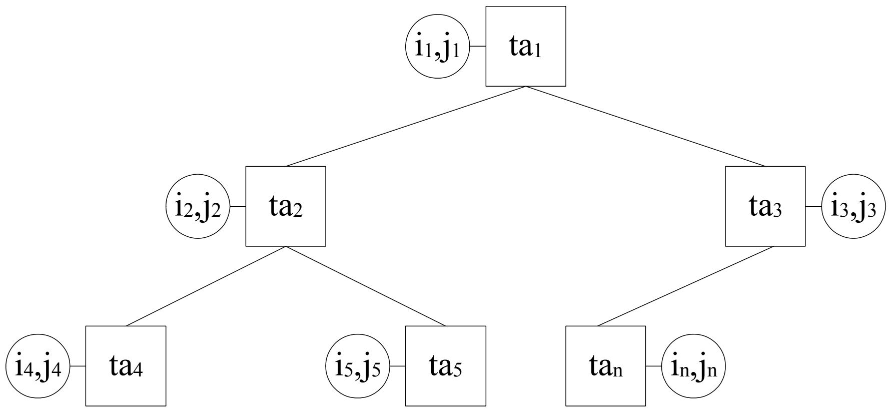


Figure 5.3: Binary heap structure. ($ta_x \leq ta_y$ for any child y with parent x)

Binary heaps have been attracting many researchers because of their simplicity and the ability to be implemented using arrays as shown in Figure 5.4. The first array location is for the root node where the minimum key resides. The arrows in the figure indicate the parent-child relations which can be obtained as follows:

$$Parent(x) = \lceil (x-1)/2 \rceil$$

$$LeftChild(x) = 2x+1$$

$$RightChild(x) = 2x+2$$

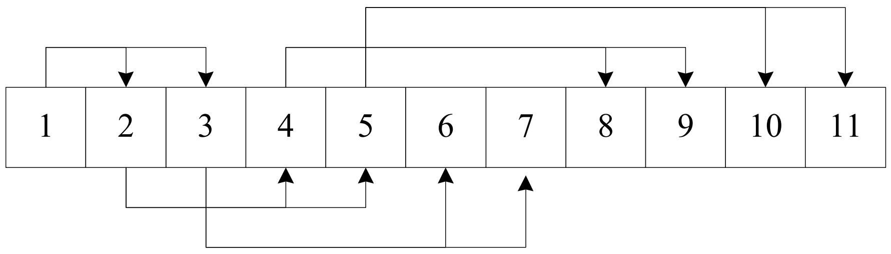


Figure 5.4: Binary heap as an array.

Since the heap is not fully sorted, the idea of combining similar time schedules in one node is infeasible as we discussed in the previous section. Therefore, redundancy in key values is allowed in a binary heap and in case the event list is requested to extract the cells with the minimum key, it should go through multiple iterations to gather all of them. This makes the method *getMinList()* of *O(n)* complexity while *getMin()* of *O(1)* since the minimum key is at the root. Removing the root node from a binary heap (i.e. remove the node with minimum key) is done by replacing that node with the last one added to the structure and then use the operation *pushDown()* to restore the heap order. The push operations *pushDown()* and *pushUp()* are done through multiple swap operations that exchange the parent node with its child in case the heap order is violated. The swap operations are repeated up or down the heap until the order is restored. Therefore, removing the root node will be of *O(log n)* since it just replaces the node with the last leaf in *O(1)* and then restores the heap order in *O(log n)*. The worst case scenario is when all the nodes in the heap are having the minimum key. As a result, *removeMin()* operation complexity is of *O(n log n)*. Finally, using the cell-location indexing scheme improves the complexity of add and arbitrary remove operations to *O(log n)*.

5.4 Binary Tree Implementation

The other form for binary search data structures considered in this work is the binary search tree which gives *O(log n)* search operation [48, 49]. In ordered to guarantee that search bound, the binary tree should be balanced, otherwise the bound will jump to *O(n)*. One of the common balanced binary trees available in the literature is the AVL tree, named after its inventors *Adelson-Vielskii* and *Landis* [50], which gives best *O(log n)* operations compared to other tree algorithms in event list implementations [51].

Binary tree keeps all the nodes sorted in non-decreasing order from left to right which guarantees that left child of any parent has a key that is less than or equal to the key of its right child. Since the tree is fully sorted, different cells can be combined in one node if they have the same scheduled time as shown in Figure 5.5. AVL tree is a height-balanced tree. It keeps the difference in height between any two childs of a node less than or equal to one. The height of a node is the maximum among its two children's heights. In a balanced AVL tree, add and remove operations may cause imbalance in the tree. The design of an AVL tree involves some rotation operations to restore balance after each add or remove operation. There are single rotate as well as double rotate operations which can be in the right or left direction depending on the imbalance situation. It was proved that these operations do not increase the *O(log n)* complexity of the *add* and *remove* operations [50].

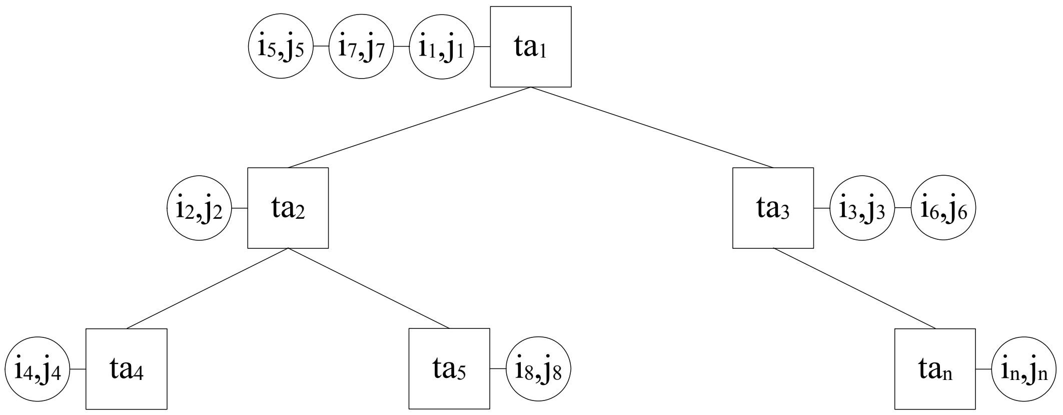


Figure 5.5: Balanced binary search tree. ($ta_4 < ta_2 < ta_1 < ta_3 < ta_n$)

The implementation of AVL tree is usually done by defining the nodes as objects and each node has two nodes as right and left childs. The cell-location indexing scheme cannot be introduced in this structure since there is no physical address for nodes as is the case in the array implementations. Instead, a cell-time indexing can be introduced to make the complexity of the arbitrary remove *O(log n)*. Following that scheme, when a cell needs to be deleted, the index will give the scheduled time of that cell and then the structure uses the *log n* search operation to browse from the root to the node containing that cell. If that specific cell is the only one in that node, the node should be deleted and rotation operations might be needed in case this deletion caused an imbalance. If multiple cells exist in that node, then only that specific cell entry is required to be deleted and the balance is still ensured. However, the worst case scenario might occur when all *N* cells exist in one node. Then deleting a single node will require a search in the internal list of that node which will make the complexity of the arbitrary remove operation *O(N)*. As a result, the complexity of the add operation will be also *O(N)* since it might remove a cell before rescheduling it. In AVL tree, the minimum time value is always in the leftmost node. In order to get that values using *getMin()*, the algorithm requires *O(log n)* operations from the root to reach that minimum value and similarly for *getMinList()*. The r*emoveMin()* method includes the *advanceTime(t)* method which costs *O(n)* and hence the *removeMin* will be in *O(n)* as well.

5.5 Analytical Comparison

Table 5.1 summarizes all complexity measures (i.e. worst case scenario) of the event list structures considered above. These figures are characterized by type of operation, type of data structure, number of entries in the event list as well as whether an indexing scheme is used or not. The fully sorted structures like the sorted array and the binary tree have different interpretations for entries. In these two structures, a list has two terminologies: records and nodes. Each node may contain more than one scheduled time record of the same time for different cells. The number of actual scheduled records is referred to as *N* which also represents the number of cells in the list while the number of the nodes in the list is *n*. On the other hand, unsorted array and binary heap structures have those numbers equal.

	add	remove	getMin	getMinList	removeMin	advanceTime
Unsorted array	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Indexed unsorted array	*O(1)*	*O(1)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Sorted array	*O(N)*	*O(N)*	*O(1)*	*O(1)*	*O(n)*	*O(n)*
Indexed sorted array	*O(N)*	*O(N)*	*O(1)*	*O(1)*	*O(n)*	*O(n)*
Binary heap	*O(n)*	*O(n)*	*O(1)*	*O(n)*	*O(n log n)*	*O(n)*
Indexed binary heap	*O(log n)*	*O(log n)*	*O(1)*	*O(n)*	*O(n log n)*	*O(n)*
AVL tree	*O(N)*	*O(N)*	*O(log n)*	*O(log n)*	*O(n)*	*O(n)*
Indexed AVL tree	*O(N)*	*O(N)*	*O(log n)*	*O(log n)*	*O(n)*	*O(n)*

Table 5.1: Complexity comparison of all introduced structures.
($n \leq N$ in sorted structures, otherwise n = N)

The worst complexity in the table is for the binary heap in *removeMin()* operation. That was estimated for the number of iterations required by the structure when all the *n* records in the heap are having the minimum value. The first one will be removed in *O(log n)*, then the others will be in *O(log n-1)*, *O(log n-2)* …etc which is summed up as follows:

$$\sum_{k=1}^{n} \log(k) = \log(n!) \le n \log n \, .$$

However, that is too far upper bound for the operation since that worst case is extremely rare in discrete event models and the algorithm might work usually close to the *O(log n)* lower bound. Therefore, that figure cannot be used to reject the idea of using binary heaps. On the other hand, the last two operations involve *O(n)* complexity inherited form the *advanceTime()* which is included also in the *removeMin()* operation. That is a tight bound which means that the operation requires exactly *n* iterations. In those operations, the advantage will be for the sorted structures in case n<N since the complexity in the unsorted ones is *O(n)=O(N)*.

Sorted array and binary heap structures outperformed all other structures in the *getMin()* operation and their runner up is the binary tree. Since sorted array has the list of the cells with minimum time in the last array location, the *getMinList()* operation requires *O(1)* operation to extract that list as an object which makes it superior in complexity to all other structures. However, these two operations are not as frequently used as the add and remove operations in simulating a large scale cellular models. Add and arbitrary remove operations are having the same complexity in each row since any add operation might include a remove operation. The remove operation is required to avoid redundancy of cells in the list. In case a cell is requested to be scheduled, the list will check if the cell already exists in the list and, if exists, remove it and then add its new schedule. Generally speaking, complexity of add operation can be improved by making sure that a cell does not exist before its new reschedule. This will reduce the complexity of the add operation as an independent operation but on the other hand requires another remove operation before the add operation which results in the same overall complexity.

As mentioned in 3.3.1, elapsed time was decided not to be implemented in the event list. However, we include analyzing the *getHistoy(i,j)* operation here to justify the previously made decision. This method can be designed in two ways. The first one is by including the history as a value in the scheduled record. The other one is by having an independent 2-D array inside the list that can store elapsed time for all cells possible in the model which is indexed based on the cell IDs. The second one is identical to the decided solution since a similar history array is used in the model and its complexity will be *O(1)* for accessing the array value. In the first way, indexed unsorted structures also give a complexity of *O(1)* while the non-indexed unsorted structures gives *O(n)* and the sorted ones give *O(N)* regardless of the indexing. Therefore, the decision, in 3.3.1, is justified by stating that the array method gives *O(1)* complexity with the advantage that outside the event list the array will count the elapsed time for all cells including the passive ones which are not considered inside the event list.

The average time spent in the event list in each iteration can be calculated as follows: $AvgTime = t_a n_a + t_r n_r + t_m n_m + t_{ml} n_{ml} + t_{rm} n_{rm} + t_{at} n_{at}$, where t_a, t_r, t_m, t_{ml}, t_{rm}, and t_{at} are the average execution time for a single call to the operations *add*, *remove*, *getMin*, *getMinList*, *removeMinList*, and *advanceTime* respectively. Similarly, n_a, n_r, n_m, n_{ml}, n_{rm}, and n_{at} are the average counts of these operations per iteration. Therefore, the total execution time spent in handling the event list depends on the distribution of the operation calls in addition to the execution time of each operation that was discussed above. This distribution is the major decision factor on selecting the best implementation which is an application dependent. The atomic model design of cellular space models that was formulated in chapter 3 allows the models to call each of the *getMin*, *getMinList*, and *removeMinList* operations once per iteration which end up with an average execution time $AvgTime = t_a n_a + t_r n_r + t_m + t_{ml} + t_{rm} + t_{at} n_{at}$. Since the *advanceTime* operation has the tight complexity of *O(n)* in all implementations, it is not a decision factor in selecting the minimum total execution time. Therefore, the relative average time that we can use to compare implementations will be $(t_a n_a + t_r n_r + t_m + t_{ml} + t_{rm})$ in which the decision factor of calls distribution only depends on the average numbers of calling *add* and *remove* operations (n_a and n_r).

Operation	Game of Life Model	Quantized Landslide Model
add	6918	6876
remove	21968	4551
getMin	100	100
getMinList	100	100
removeMin	100	100
advanceTime	0	0

Table 5.2: Operation counts in two models for 100 iterations.

Add and remove operations are the most frequently called operations in large scale cellular space models since they contain a huge number of cells that are required to be added or removed from the list. Table 5.2 gives a quick insight on operation counts in running two models that represent discrete time and discrete event approaches. The run was done for a 32×32 cell space that was entirely encapsulated in an atomic model which does not receive external messages and this explains not calling the *advanceTime* operation explicitly. The *add* and *remove* operations were shown to be the most frequent compared to the others. Therefore, the structures with the fastest add and remove operations, will most likely be the best candidates for event list implementation in our system. In Table 5.1, it is clear that the structure which satisfies this is the indexed unsorted array structure and its runner up is the indexed binary heap.

5.6 Experimental Analysis

Table 5.1 shows that the indexed scheme gave an advantage over the non-indexed implementations in add and remove operations. In this section, we perform experimental analysis for those indexed structures. Four event lists were implemented, one for each of the above mentioned structures. These experiments were designed to give a quick guide in making a decision on which structure to select for implementation. All runs were done on a windows XP machine having a Pentium 4 processor with 3.0 GHz speed and 1 GB RAM. The execution time was measured using the Java facility of retrieving the system time in milliseconds. In the add operation, the time was measured for all add operations done for an empty list to make it reach the specified size and this is done backward for the remove operation. All other operation's time was measured for one method call at that list specified size.

The first part targeted the large scale list implementations. In this part, lists of 10000, 50000 and 100000 records were run to compare the execution time for each operation. Tables 5.3 through 5.5 list the results of the three runs done in this part. The table records with zero seconds means that the execution time was less than 0.5 milliseconds. The three tables gave an indication that as the number of records gets extremely large, the unsorted array and the binary heap become advantageous. The worst performance was for the AVL tree which when it gets very large, it has very large objects to deal with. This requires huge memory allocation and if not available, more time will be wasted in memory misses and caching. Another reason is that balancing the tree in add operations seems to be very costly since they are much worse than other operations. As a conclusion of this part, keeping the list fully sorted is very costly in case of the add operation. Since large scale cell spaces are targeted in this work, the AVL and the sorted array might not be of interest due to their poor performance in very large scale lists.

	Add	Remove	getMin	getMinList	removeMin	advanceTime
Unsorted array	0.02	0	0	0	0.02	0.02
Sorted array	0.09	0.02	0	0	0	0.06
Binary heap	0.05	0.02	0	0	0	0.02
AVL tree	0.61	0	0	0	0.02	0.02

Table 5.3: Execution time in seconds for lists with 10000 records.

	Add	Remove	getMin	getMinList	removeMin	advanceTime
Unsorted array	0.05	0	0	0.03	0.05	0.05
Sorted array	2.22	0.02	0	0	0.05	0.84
Binary heap	0.2	0.05	0	0.03	0.03	0.05
AVL tree	327.92	0.02	0	0	0.97	0.92

Table 5.4: Execution time in seconds for lists with 50000 records.

	Add	Remove	getMin	getMinList	removeMin	advanceTime
Unsorted array	0.08	0.02	0	0.16	0.09	0.08
Sorted array	11.53	0.03	0	0	0.17	3.45
Binary heap	0.42	0.09	0	0.14	0.06	0.09
AVL tree	2821.39	0.02	0	0	3.92	3.73

Table 5.5: Execution time in seconds for lists with 100000 records.

The second part of the experiments targeted the two structures qualified from the first part. In that part, we can conclude that in large scale lists, the unsorted array structure outperformed the binary heap in the *add* and *remove* operations while the opposite is true in the *getMinList* and the *removeMin* operations. The second part targeted the small size lists with unsorted array and binary heap structures. Two experiments were run: (a) records having big number of time redundancies; (b) with less or almost no redundancies. The aim behind this is to test both lists in two extremes. These tests might differentiate between the two lists on a performance basis. Tables 5.6 through 5.10 show the execution times of all operations in microseconds for both structures in experiments a as well as b. A clear advantage of using the unsorted array in the *remove* operation while in the *add* and the *removeMin* operations the difference is negligible. On the other hand, the binary heap outperformed the array in the *getMin* and the *getMinList* operations.

	n=10	n=100	n=1000
Unsorted Array-a	9	44	389
Binary Heap-a	13	45	394
Unsorted Array-b	11	45	405
Binary Heap-b	11	45	409

Table 5.6: Execution time in micro seconds in *add* operation.

	n=10	n=100	n=1000
Unsorted Array-a	2	19	209
Binary Heap-a	18	138	1998
Unsorted Array-b	3	19	208
Binary Heap-b	22	141	2004

Table 5.7: Execution time in micro seconds in *remove* operation.

	n=10	n=100	n=1000
Unsorted Array-a	3	5	42
Binary Heap-a	0	0	0
Unsorted Array-b	2	3	33
Binary Heap-b	0	0	0

Table 5.8: Execution time in micro seconds in *getMin* operation.

	n=10	n=100	n=1000
Unsorted Array-a	6	16	136
Binary Heap-a	5	5	20
Unsorted Array-b	3	11	114
Binary Heap-b	2	2	2

Table 5.9: Execution time in micro seconds in *getMinList* operation.

	n=10	n=100	n=1000
Unsorted Array-a	3	5	9
Binary Heap-a	3	6	11
Unsorted Array-b	13	42	411
Binary Heap-b	13	41	386

Table 5.10: Execution time in micro seconds in *removeMin* operation.

5.7 Concluding Remarks

As discussed in the above sections, each of the event list's implementation strategies has its own strengths and weaknesses. Some of them perform well in small sized lists and worse in large scale ones. Others might perform well in all list sizes in some operations while they are not as good as in other operations. Selecting the best among them is application dependent. The first decision factor is if the list size is a major concern for that specific application. Then, a decision can be made based on the distribution of operations in the application, which means how frequently each of the list operations will be used during the execution.

In this work, the system under design which is in need of event list implementation is an atomic cell space DEVS model. The scalability issue is one of the major design issues in cell space models. Therefore, event lists in these models must be among the ones that perform very well in handling a huge number of records. This is why the sorted array and the binary tree implementations were skipped in the second part of the experimental analysis. Experiments in that part show that the unsorted array is good in the *add* and *remove* operations while the binary heap is good in the *getMin* and *getMinList* operations. The cell space atomic DEVS model, as formulated in section 3.3, deals with the event list mainly inside the internal transition function. The operation *advanceTime* is the only one called in the external transition function and it is only called once when receiving external messages. In each internal transition, each of the *getMin*, *getMinList* and *removeMin* operations is called exactly once while the *add* and *remove* calls depends on the number of active cells in the space. Therefore, in very large scale cell spaces the number of active cells will be very high which will result in huge numbers of add/remove calls compared to other calls and this will give advantage to the structure with extremely high *add* and *remove* operations. The only structure with very fast *O(1)* *add* and *remove* operations is the indexed unsorted array structure and this is the one selected to be implemented in our system. However, in case another structure shows better performance, it can replace the existing one without modifying the internal system given that the new list should be implemented following the design requirements listed at the beginning of this chapter.

CHAPTER 6 : TESTING AND VERIFICATION

This chapter is devoted to describing the procedures and the proposed methods followed to test, validate and verify our modeling tool as well as its generated models. The major test intended in this work is to prove that the generated models, based on the new proposed formalism, are identical to the conventional cellular DEVS models. Before getting into that concluding test, the implemented modeling tool should go through some classical software testing techniques in order to guarantee its correctness. In addition, the generated models should go through some code tests and then model validation and verification procedures.

6.1 A Quick Overview on Software Testing Techniques

Most of the software testing literatures [52-58] present testing as an integrated process within the overall software development process. The V-model is frequently used, e.g. [53, 54], to explain the need of testing for each phase of the software development process. Since 100% error-coverage is impossible in testing, a large number of different techniques were introduced to uncover the largest possible percentage of the errors. Many efforts (for example see [53, 54, 56]) were done to gather and classify the most common techniques available for testing practitioners.

Generally speaking, testing techniques can be static or dynamic. Static techniques involve the ones that can be done without running the System Under Test (SUT). This can include the design reviews, checking manuals, as well as testing hardware and/or software requirements to run the SUT. On the other hand, dynamic techniques, which can be implicit or explicit, are used while running the SUT. The most common type of testing is the explicit dynamic techniques that run test cases to check if the SUT produced the expected results for each case [59]. Test cases can be designed in different levels of the software development namely the unit testing, integration testing, system testing, and acceptance testing levels. The approaches used in designing test cases can be of two types: black box testing or white box testing. In contrast to the white box approaches, black box approaches are designed with no knowledge regarding the internal design of the box (i.e. SUT). The gray box approach was introduced in [54], which assumes a prior knowledge of the implementation but the box is then closed in order to design more effective black box test cases. The challenge that every designer will face is to determine the smallest number of test cases that can be run to check the correctness of the largest number of system states as covering all states and paths is impossible [56].

6.2 A Quick Overview on Simulation Model Validation and Verification

The most important questions that a model designer should answer after developing a model are: is the model correct?, and is that developed model the right one? Usually, answering the first question is referred to as verification while to the other one is validation. Similar definitions can be found in the literature with different details and constraints [6, 60-62]. Once a model is built, it has to be validated in order to check if it is generating the intended system behavior based on the given conditions. This is done through comparing the input-output transformation of the model to the system's input-output transformation. On the other hand, verification is the process of checking how accurately the model represents the conceptual model, which is a transformation equivalency check. Figure 6.1 overviews the model development life cycle and explains the validation and verification relations. Verification and validation techniques are overviewed and classified in [60] as informal, static, dynamic, symbolic, constraint, and formal techniques.

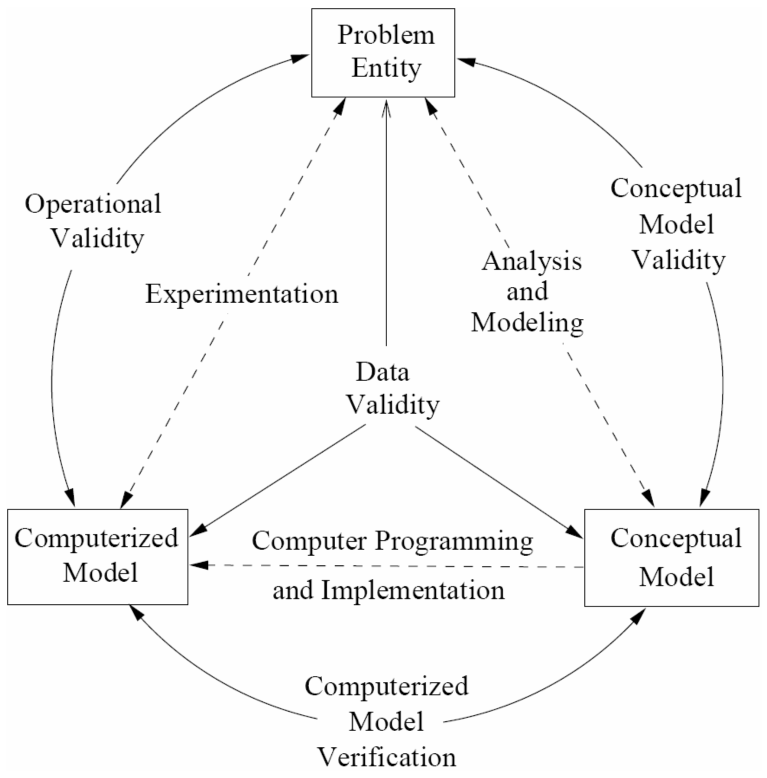


Figure 6.1: Simplified modeling process. Taken from [61]

6.3 Validation and Verification of DEVS Models

According to [6], verification in DEVS framework is to check if the simulator is in error while validation is to check if the model is in error. Two general approaches of simulator verification were mentioned: the formal proof of correctness, and the extensive simulator testing. The second one is the most attractive one for researchers since complete formal proofs are absent. There are very few recent efforts done to obtain partial mathematical proofs with some constraints [63, 64]. Similarly, in validating DEVS models, the most attractive techniques are the simulation based ones. Among them, the most common one is the experimental frame strategy (i.e. generator/transducer approach) [6, 65]. In this approach, the model is treated as a black box that is tested using an experimental frame. The frame includes a generator that generates inputs to the model and provides the transducer with the output values it should expect from the model. The transducer then compares the outputs generated by the model with the expected values and validates the model accordingly. Based on the generator/transducer approach, [66] presented an automated DEVS model verification process which is essential in reducing the time and cost of testing. However, that implemented approach does not support verifying models that might change states in zero time. In this work we propose an extension to that work in order to support all DEVS models.

6.4 Testing the Cellular DEVS Specification Unit

The main unit in the structured design we presented in Figure 4.2 is the cellular DEVS specification unit. It contains all the generic classes and methods required by any cellular DEVS model that will be generated. Testing this unit is essential since it is responsible for guaranteeing the anticipated high performance as well as the correctness of this work. This unit is divided into different classes and interfaces as shown in Figure 4.3. The main classes that we are interested in testing are: *cell*, *block*, *cellSpace*, and *blockSpace* classes. The inheritance implemented for these classes allows us to test all methods in parent classes. By testing all classes, constructors and methods, we guarantee that the unit precisely constructs and initializes the models. This involves adding and labeling input and output ports as well as coupling all internal and boundary ports in case of the *cellSpace* or *blockSpace* Classes.

6.4.1 Testing *cell* and *block* Classes

These classes were intentionally designed to have no behavior since it is usually specified in the generated models that extend these classes. Therefore, the designed test cases are employed to test the constructors for correct initialization and check the correctness of the methods. Two test cases were designed to check the empty as well as the non-empty constructors. The methods in the *cell* as well as the *block* classes can be classified into two different types. The first one is the methods that return values while the other one include the methods that do not return values but change the class state. Examples of the first type includes: *portDir*, *oppositeDir*, *userNeighbor*. Assertion like methods, which are referred to as black box methods in software testing, were used to design the test cases which check if a method returns the correct values according to the given inputs and conditions.

On the other hand, the *addPorts* methods cannot be tested using the assertion. Instead, test cases were designed to check if the model state changes correctly. After calling the methods that add ports, test cases should check if the correct number of the ports were added to the cell or block. In addition, they should check if the port names fall correctly with in the labeling scheme followed in chapter 4. This was done with the help of regular expressions in Java. Table 6.1 and Table 6.2 list all the regular expressions used in testing the generated port labels for cells and blocks. In addition, the tables list the number factors that are used by the test cases in calculating the expected number of ports to be added to the class. For example, if an N×M block (with both N and M greater than 2) has 3 single ports and 2 multi-ports with the Moore neighboring rule, the expected total number of ports is 3×(2N+2M-4) + 2×(6N+6M-4). Table 6.2 shows more figures in the case of single ports in the block. This reflects more details on how the addressing was handled to support one dimensional blocks as well as the blocks that only have one cell.

	Regular Expression	Ports Num
Single port	[^_]+	1
Neumann multi-port	[^_]+_(N|S|W|E)$	4
Hex multi-port	[^_]+_(W|E|NW|SW|NE|SE)$	6
Moore multi-port	[^_]+_(N|S|W|E|NW|SW|NE|SE)$	8

Table 6.1: Regular expressions and ports number factor in cell test.

	Regular Expression	Ports Num
Single port – 1 cell	[^_]+	1
Single port – vertical 1D	[^_]+_(S|N|([0-9]+))$	N
Single port – horizontal 1D	[^_]+_(E|W|([0-9]+))$	M
Single port – 2D	[^_]+_(SE|SW|NW|NE|((N|S|W|E)_([0-9]+)))$	2N+2M-4
Neumann multi-port	[^_]+_((N|S|W|E)_([0-9]+))$	2N+2M
Hex multi-port	[^_]+_((N|S|W|E)_([0-9]+)_(W|E|NW|SW|NE|SE))$	4N+4M-2
Moore multi-port	[^_]+_((N|S|W|E)_([0-9]+)_(N|S|W|E|NW|SW|NE|SE))$	6N+6M-4

Table 6.2: Regular expressions and ports number factor in block test.

6.4.2 Testing *cellSpace* and *blockSpace* Classes

In testing the *cellSpace* and *blockSpace* classes, similar test cases were designed for the constructor test and *addPorts* methods. Regular expressions format and port number calculations follow in Table 6.2 in testing the *addPorts* method. The most critical methods that play essential roles in the coupled space behavior are the ports coupling methods. These were divided into two types: internal couplings and boundary couplings. Internal methods generate the necessary coupling between cells so that they can communicate with their neighboring cells. The boundary coupling methods make the coupling of the boundary cell to the space ports so that they can communicate with cells in other neighboring spaces. Getting the coupling list in a test case is made possible using the *getCouprel* method in DEVSJAVA. The test cases were designed to check if the number of couplings generated is correct or not. There are many formulas that were obtained for the test cases to calculate these numbers based on the number of input/output ports, port types, and neighboring rule followed. In addition, the test cases are responsible for checking if the port names in the couplings are correct and following the specific port labeling using the regular expressions of Table 6.1 and Table 6.2.

6.5 Testing the Generated Models

This is an essential test that is required to be run every time a new model is generated by our development tool. Seeking user convenience as well as testing time reduction, the test cases were designed with support of automation. There are a number of classes that are designed and made available for the user to call and test the generated model. These tests include two types: generated code test and atomic cell DEVS verification.

6.5.1 Generated Code Test

This is a quick test that checks that the automatic code generator works correctly. It makes sure that the generated code is correct Java source code, and a complete DEVS cell space model. This is done through Java reflection to check that the code has the correct classes, constructors, methods, and variables according to the design of the code generator.

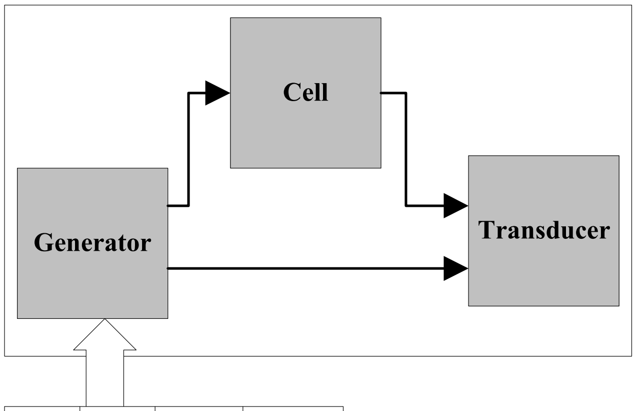
The test starts by searching for the generated model classes using their given names. It returns an exception if the class cannot be found. This either means that no such model was generated with those names, or the model code has some syntax errors and its class file was not generated. The first procedure, which is done by the command *Class.forName("UnitName")*, makes sure that the generated code is correct Java code. The second one, presented in the first block of Figure 6.2, makes sure that the generated classes are cellular DEVS models that are extending the generic classes in the cellular DEVS specification unit. This is done by the reflection method *getSuperClass()* which returns the parent class of the class under test. The next two blocks check that the generated classes have the correct number of variables and constructors using the methods *getDeclaredFields()* and *getDeclaredConstructors()*. The last one checks the number of methods generated in the model and makes sure that it has all principal methods required for a standard cellular DEVS model.

```
Class cellClass = Class.forName(unitName);

Class sup=cellClass.getSuperclass();

if (!sup.getName().equals("newCellDEVS.cell")
&& !sup.getName().equals("newCellDEVS.block")) return false;

Field[] fld=cellClass.getDeclaredFields();

if (fld.length<2) return false;

Constructor[] cons=cellClass.getDeclaredConstructors();

if (cons.length!=2) return false;

Method[] meth=cellClass.getDeclaredMethods();

if (meth.length<8) {return false;}
else {
   check=false;

   for (int i=0; i<meth.length; i++){
      if (meth[i].getName().equals("deltext")) check=true;

   }
   if (!check) return false;
   check=false;
   for (int i=0; i<meth.length; i++){
      if (meth[i].getName().equals("deltint")) check=true;
   }
   if (!check) return false;
   check=false;
   .
   .
}
```

Figure 6.2: Examples of using Java reflection in testing unit class.

6.5.2 Atomic Cell DEVS Verification

The above test does not guarantee that the model is right. To do so, it should be run and compared to some verification data provided by the model developer. [66] presented an automatic verification framework for cellular DEVS models which is similar to the experimental frame idea presented in [6]. Testing the cell as an independent unit was proposed to verify the cell behavior according to the given verification data. This kind of test is referred to as a unit test in software literature [53-55]. It follows the black box idea by having a generator that inputs data to the cell and the output of the cell is then monitored by a transducer which checks those values with the expected ones provided by the generator. Based on that, we introduced an automated way of verifying generated cells in our development tool as shown in Figure 6.3.



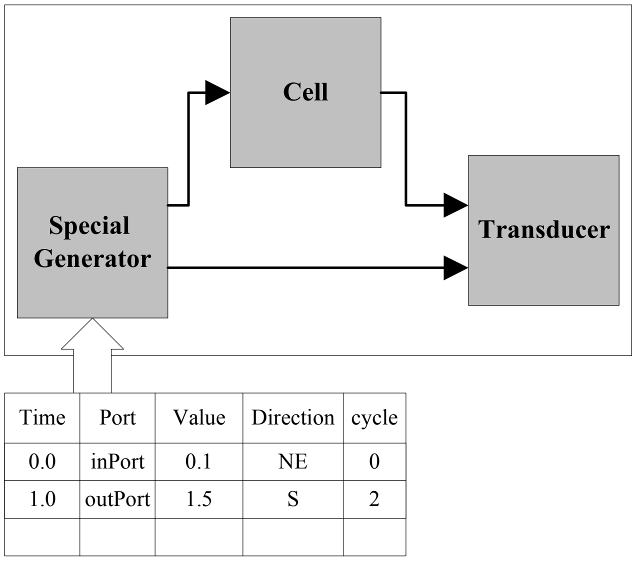
Time	Port	Value	Direction
0.0	inPort	0.1	NE
1.0	outPort	1.5	S

Figure 6.3: Cell verification experimental frame.

The verification class, represented by the white box in Figure 6.3, was designed as a generic DEVS coupled model that can be used by any developer. It supports an automated construction of the generator and the transducer based on the provided cell under test (CUT). This is done by recognizing the list of input/output ports of CUT and then generating the list of ports for the generator as well as the transducer. If a cell has M input ports and N output ports, the generator should have M+N output ports and the transducer will have 2N input ports. The couplings are done as shown in Figure 6.3.

The verification data is read by the verification framework from an external file which should be in the format shown the figure. Each record should have a time stamp that specifies when the event will occur, which port will handle the event, the testing value, and the direction of flow in case of multi-ports. The data is processed to produce the corresponding port labeling (section 4.3.2), which is always kept transparent to the user, and to classify the type of data to be used for each port. Then it is fed to the generator to start the verification process. A record that includes an input port name indicates that the test data should be sent to the CUT while the one with output ports is interpreted as the expected output value and is sent to the transducer. The generator is held responsible to generate the verification data to the CUT as well as the transducer. The transducer, which can be seen as a comparator, checks if the CUT generated the expected values received from the generator.

The above proposed cell verification framework works fine in a wide variety of cell DEVS models especially the cellular automata and the discrete time models. Unfortunately, it cannot be successful in all cellular DEVS models, particularly the ones that allow cells to do multiple zero time state transitions. This is because the generator usually sends the expected events to the transducer upon reaching their specified time. Any output generated later in zero time will be considered as incorrect output generated by the cell and hence it erroneously fails the test. A new special generator is proposed to overcome this problem. Figure 6.4 shows the new proposed framework that uses a special generator. The whole idea is the same as above except that the generator is enhanced to be more intelligent and produce the expected output values correctly even with zero time state transitions. One more entry is added to the verification data to indicate the cycle in which the generator should produce that test value. In this case, the test is not considered a black box approach anymore since the cycle number requires prior knowledge of the internal design of the cell.



Time	Port	Value	Direction	cycle
0.0	inPort	0.1	NE	0
1.0	outPort	1.5	S	2

Figure 6.4: Cell verification experimental frame with special generator.

Verifying cells guarantees that each cell generates the required behavior if it runs independently. As a result, the correctness of the cells integration inside one cell space can be ensured if and only if all port couplings are correct. This can be guaranteed by the coupling test introduced in section 6.4.2. This does not apply for block verification. The cell rules of a block can be tested in a similar way to cell verification. In this case, a block should include one cell and then it is treated as a regular cell and fed to the verification framework to be tested. Verifying a block with multiple cells will follow the verification method in the next section.

6.6 Verifying the Approach

The multi-layer approach presented in this work is based on generating an equivalent atomic model of the conventional coupled cell space approaches. Before running experiments and making conclusions, the new generated atomic models must be made identical to the original coupled DEVS implementation. In this work, the simulation verification method is followed in order to dynamically check the equivalency of the two approaches. Figure 6.5 illustrates the idea of dynamically comparing simulation runs of two systems. A DEVS atomic model is employed to work as a comparator that monitors the state trajectories generated by both systems and generates a fail report if they are not equivalent. The report should include the non-equivalent values generated as well as their time. If no fail report is generated and the verification run comes to an end, the two systems can be declared identical. In case of non-terminating simulations, the larger number of iterations employed, more is the confidence that will be obtained in the conclusions.

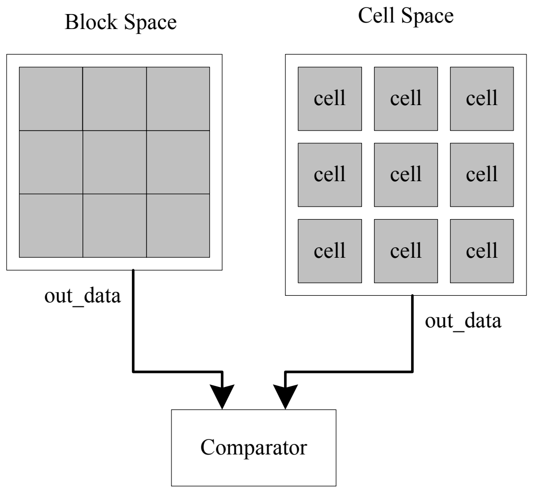


Figure 6.5: Simulation approach of equivalency verification.

6.7 Applying All Tests

All the tests explained in the above sections were applied to the new cellular DEVS development tool. The cellular DEVS specification unit tests are the only types of tests that are model independent. Those tests can be run without making a full model development phase. It was run for large number of inputs (e.g. common and extreme cases) to make sure that all classes and methods perform correctly. Figure 6.6 and Figure 6.7 show sections of the specification unit test report. The report lists all the results of test instances and concludes with the overall unit result at the end. The second part of the tests is the generated code tests and verification runs that require selecting some models to be developed and run. Three test models were selected for testing and verification. The first one is the two dimensional game of life which is know for its popularity and simplicity. The second one is the simple 2-D wall following robot which is popular in artificial intelligence texts (e.g. [67]). The last one is the basic finite difference numerical solution of a one dimensional heat equation [68].

```
================================================
Testing the cell space specification unit started...
================================================

1.   Testing  the cell class.
        constructors :   Pass
        protDirM method :   Pass
        oppositeDir method :   Pass
        userNeighbor methods :   Pass
            addSinglePorts :   Pass
            addNeumannPorts :   Pass
            addHexPorts :   Pass
            addMoorePorts :   Pass
            addAllPorts :   Pass
        addPorts methods :   Pass
     Cell Test Result: Pass
----------------------------------
```

Figure 6.6: Beginning of specification unit test result.

```
4.   Testing  the block space class.
            empty constructor :   Pass
            non-empty constructor :   Pass
        constructors :   Pass
            addSinglePorts :   Pass
            addNeumannPorts :   Pass
            addHexPorts :   Pass
            addMoorePorts :   Pass
            addAllPorts :   Pass
        addPorts methods :   Pass
        Boundary Couplings methods :   Pass
        Internal Couplings methods :   Pass
     BLock Space Test Result: Pass
----------------------------------

Test Result of the cell DEVS specification unit:   Pass
==============================================
```

Figure 6.7: End of specification unit test.

These three models span wide varieties of variable types (i.e. state vs. flow variables), data types, boundary values, as well as discrete time versus quantization based approaches. The wall following robot model is selected to show the support of our environment for modeling propagation and flows between cells. It is also considered as an extreme case model test since all cell activities are mostly at the boundaries. It uses the Neumann neighboring rules where the game of life uses the Moore neighboring rule. The heat equation solution employs the quantization approach which is representing a large number of applications that are based on differential as well as partial differential equations. This is considered also as an extreme case test since the environment was designed originally for two dimensional cell spaces. Passing the tests will conclude that the environment is capable of developing 1-D cell spaces correctly as well.

All testing methods were applied to these three test models. First of all, the development environment was used to produce the models through the GUI. Then, the generated models were run through the Java reflection test in order to ensure the correctness of the code. This was done by writing a simple code to call the code testing class (section 6.5.1) and send it the model name to be tested. A similar approach will be used in calling the cell verification test. However, instead of sending the model name, the user is required to construct and initiate the cell and send it to the *verifyCell* class provided with the verification data file name. By running the main method in that class, the verification will start and check that the cell behavior is correct according to what the user intended to design. One further step is done here by verifying the block rules as well. This can be done by initiating a block that contains one cell and verifies it with the normal cell verification procedures. The last test is the equivalency verification of the block space to the cell space. All possible blocking (i.e. cell encapsulation) setups were constructed and verified to be equivalent to the original cell space models.

Figures 6.8 through 6.10 show some examples of the test reports generated when verifying cells, blocks, and spaces. Figure 6.8 lists two cell verification runs. The first one is for a game of life cell that passed the verification test since the simulation terminated with no errors output. The other one is for a non-passing block that contains one game of life cell. The output errors stated that according to the verification data, the cell generated wrong values at time 5.0 and did not produce an output at time 8.0. This kind of information guides the developer in debugging the generated model by looking for those rules and places that might be the cause of those specific errors.

```
a.  cell verification
     ---Now, verifying the unit atomic model..
     Terminated Normally at ITERATION 29 ,time: Infinity
     ---If no error printed above this line, the unit is verified

b.  block verification
     ---Now, verifying the unit atomic model..
      Time: 5.0 Outport [outLife]  produced 1, expected value is: 2
      Time: 8.0 Outport [outLife]  produced  0  , but not supposed to
     Terminated Normally at ITERATION 29 ,time: Infinity
     ---If no error printed above this line, the unit is verified
```

Figure 6.8: An example of cell verification test report.

Figure 6.9 and Figure 6.10 show reports on the equivalency verification of the approach for the heat equation model. The report should list any non-equivalency between the cell space and the block space. Figure 6.9 represents a fail report that concluded that the two cell spaces are not identical. In addition, it lists the unmatched values and the time of mismatch occurrence. On the other hand, Figure 6.10 illustrates a verification pass report for the wall following robot. Different blocking setups were tested and all verification runs terminated successfully with no single error.

```
blocking: 1 , 1
Model-1[99][0]=280.0 Model-2[99][0]= 270.0 at time : 354.557
FAIL:  The two models did not pass the test. [They are not identical]
Model-1[99][0]=290.0 Model-2[99][0]= 280.0 at time : 374.493
FAIL:  The two models did not pass the test. [They are not identical]
Model-1[99][0]=300.0 Model-2[99][0]= 290.0 at time : 395.379
FAIL:  The two models did not pass the test. [They are not identical]
Model-1[99][0]=310.0 Model-2[99][0]= 300.0 at time : 417.309
FAIL:  The two models did not pass the test. [They are not identical]
```

Figure 6.9: An example of non-identical spaces.

```
blocking: 1 , 1
Terminated Normally at ITERATION 1001 ,time: 200.0
 blocking: 1 , 2
Terminated Normally at ITERATION 1001 ,time: 200.0
 blocking: 1 , 4
Terminated Normally at ITERATION 1001 ,time: 200.0
```

Figure 6.10: An example of identical spaces.

CHAPTER 7 : LANDSLIDE APPLICATION MODELS

The work presented in this chapter illustrates the capability of developing complex natural models using the new environment. Different landslide models were developed to show the support of the environment to different modeling abstractions and requirements as well as the expansion possibility to make it more generic. At the end, the simulation runs of those models were used to justify the speed up gained when using the new formalism compared to the conventional cellular DEVS implementation.

7.1 Overview on Landslides and the Need for Their Models

Landslides are among the major natural hazards that occur frequently on earth. In addition to the loss of lives and infrastructure damages, they have a great impact on the land formation and evolution. Research studies in this area include: detecting, classifying, monitoring, analyzing, and predicting landslides. The ultimate goal of modeling landslides is the prediction which helps in saving lives and reducing damages. The complexity nature of such models as well as the large list of involved factors made it extremely hard to agree on one universal model. Many models were proposed in the literature [69-79] spanning different strategies and landslide factors. One of the major factors triggering landslides is the slope failure which is caused by other factors like rainfalls, earthquakes, and soil mechanics.

Landslide models that are based on slope failure (e.g. [70, 75-77]) calculate the local slope of a land at each section and decide on its criticality. On slope failure criticality, a land section triggers a local slide to get back to a non-critical state based on the self-organized criticality [74, 80-83] nature of the landslides. A local slope failure in a land section might cause failures in other sections and the integration of all failures forms a global landslide. The differences in landslide models were also extended into the mass flow equations. Different researchers derived different equations that generate the debris flow behavior during a landslide. The complexity and non-linearity of these partial differential equations require solving them using numerical methods. These approaches involve discretizing the land surface spatially into two or three dimensions. The most attractive way in the last decade is by using cellular automata in simulating natural physical systems (e.g. [37, 38, 70, 71, 77, 84, 84-87]).

7.2 Cellular DEVS Models for Landslides

The cellular space modeling approach divides space into discrete cells where local computations held in each cell are based on its own as well as its neighbor's states. The neighboring rules are obtained based on the lattice setup which, in our environment, might follow Neumann, Moore, or hexagonal neighboring rules as shown in Figure 4.4. Landslide models presented in this chapter are all following hexagonal neighboring rules in which each cell is represented as a hexagon that is surrounded with six neighboring hexagons which are numbered according to Figure 4.5c. The selected landslide models are based on the models derived in [70] and [77]. The first one is a pure cellular automata model since the time plays no role in all equations while the other one is a discrete time cellular automata model that is solving partial differential equations. Two more models were introduced based on the second approach. The third one is derived by using the quantization scheme that employs discrete event rather than discrete time simulation. The last one is a rate-based predictive quantized landslide model.

7.3 Non-Timed Cellular Automata Landslide Model

The landslide model presented in [70] is a pure cellular automata model that does not account for time in its flow calculations. The landscape is divided into hexagonal cells. Each one has its own state variables, parameters as well as transition function. The state variables includes: cell altitude (i.e. elevation of bedrock + depth of soil cover), thickness of landslide debris, depth of erodable soil cover, and debris outflow to all six neighbors. Figure 7.1 shows more variables that represent the cell state during calculations like the kinetic head h_k, cell height (h = bedrock elevation + erodable soil + thickness of landslide debris), and the overall run-up height r. Since all cells in a specific landslide application have fixed area A, the land mass movements are represented using heights instead of volume.

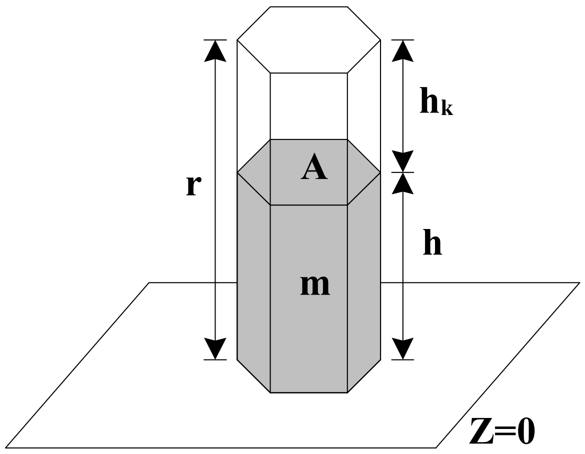


Figure 7.1: A representation of hexagonal land cell that accounts for energy using the kinetic head.

The cell transition function is composed of four elementary processes: calculating debris outflows, updating local landslide debris thickness and energy, mobilization triggering and effect, and energy loss calculations. By the end of each process all cells must be synchronized and updated with the new neighboring states. Therefore, the transition function should be implemented using four synchronized phases. In the first phase, the height is virtually incremented to r in order to account for run-up effects and then the slope angles in all directions are calculated. To minimize the critical directions, average flows are calculated and then an iterative minimization algorithm is employed to eliminate directions with flows less than the average. In the second phase, all cells are updated with the new flows which also involve energy state update. Based on the new local energy value, the soil erosion in a cell is calculated in the third phase. This process is responsible of converting some parts of the bedrock into movable soil that might move with the current debris flow which might cause a change in the local cell energy. The last phase determines the energy loss that is caused by friction which is also represented by a reduction in the run-up height. For more detailed equations and explanations see [70].

7.4 Discrete Time Cellular Landslide Model

Segre and Deangeli introduced a lattice independent cellular automata model of landslides in [77]. Instead of specifying all different directions of a cell based on the lattice setup, they came up with equations that are based on directional vectors and derivatives. The flow equations were solved using a discrete time step that is needed to be calibrated for each specific application in addition to some other parameters. Unlike the previous model, this one does not account for energy and run-up effects. It is mostly about mass conservation in which the model specifies a criticality condition that is when met, a mass flow is initiated from the higher cell into its critical direction(s). When a flow passes into a cell, a continuation factor is added to the criticality condition that might trigger flow into previously non-critical direction(s). More details were given to the soil content that represent cell state. The state variables in this model involve the bedrock height as well as the movable soil content which is composed of five variables: gas, water, silt-clay, sand-gravel and boulders. The model calculates the flow rate for each of the five soil contents using the solid content density, concentration, friction angle, and slope vectors in each of the cell's directions. Figure 7.2 lists the calculated 2-D directional vectors $\vec{J_v}$ that are originated at center cell (0,0). For simplicity, in our implementation, the slope vector is defined in one direction as $\vec{p}(i,v) = (\vec{J_v}, \Delta z_v(i)/d)$ rather than calculating the steepest descent vector between every two directions which by the end account for each direction twice. The slope vector calculation is illustrated in Figure 7.3 where the height difference in direction v is referred to by $\Delta z_v(i)$ and the distance between centers of two cells is d. All other flow equations and parameters were used as is in [77].

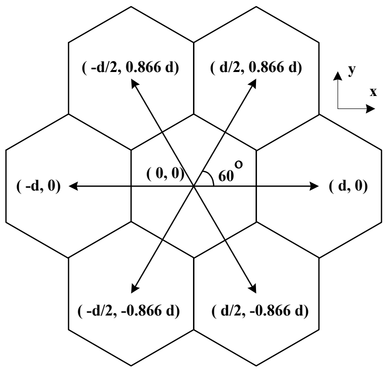


Figure 7.2: Directional vectors of cell neighbors ($\vec{J_v}$) in hexagonal lattice.

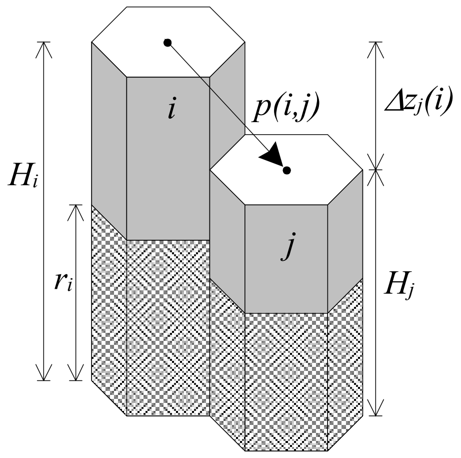


Figure 7.3: Slope vector calculation.

7.5 Discrete Event Cellular Landslide Model

The use of discrete events, rather than fixed time steps, in simulation gives a significant speedup in many applications [6]. To convert the above discrete time model into discrete event one, a quantization scheme is needed to be employed. Instead of advancing cells to the next fixed time, each cell is required to calculate its own next time advance based on a specified quantum level. The main state variable that is of a great interest in simulating landslides is the height. When a cell becomes in critical state, it starts sending mass flows into its neighbor(s). The previous model calculates the mass to be sent to all directions based on the time step. This model first calculates the rates of change in the internal content that are add up to form the rate of change in cell height. Then, defines the time step that makes the change in height exceeds the quantum level using the following equation:

$$\Delta t = \frac{quantum}{\sum_{v}\sum_{k} q_k(i,v)}$$

, where the flow rate for each of the mass contents *k* to direction *v* is:

$$q_k(i,v) = Q_k \sqrt{\rho h^3 \sin\theta(i,v)}$$

, given that ρ is the solid material density, θ is the slope angle, *h* is the cell height, and Q_k is content flow constant. After defining the time step, all mass flows are calculated and sent to neighboring cells on the expiration of that calculated time step. One modification was done in the equation by raising *h* to the power 3 instead of 5 as in the paper. The original equation causes huge flow rates and requires the time steps to be in microseconds to run correctly. The modification was done to relax the huge computational power required and to correctly simulate the flows in terms of milliseconds.

7.6 Rate-Based Predictive Quantized Landslide Model

All the previous models might produce wrong cell decisions or instability during simulation. If a cell is found to be critical and starts mass flows to its neighbors, the flows should be stopped by the time the neighbors become non critical. This stopping rule is not modeled so far and the models counts on the size of the time step to be very small to avoid passing that point. In previous runs of some landslide models, it was found that the time step must be extremely small to avoid incorrectness and instability which might degrade the simulation performance. In addition, the cells are not aware if the neighboring cells are receiving other mass flows from other cells which might cause the cell to mistakenly send flows also to those neighbors. As a result, that receiving cell might grow very fast until it becomes critical and send the flows back to originating cells. This causes a form of unrealistic and unstable runs like the example shown in Figure 7.4.

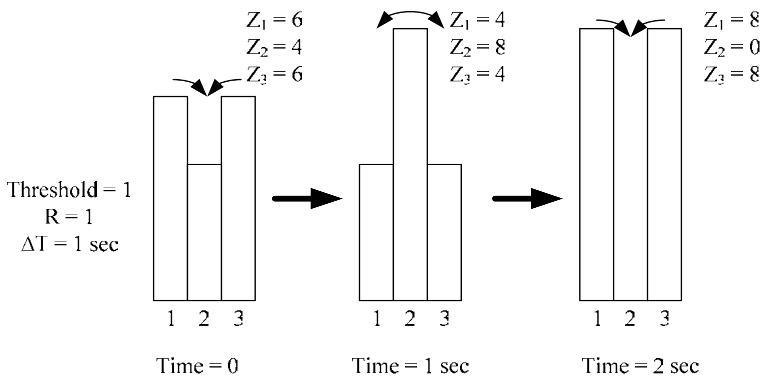


Figure 7.4: An example of incorrectness and instability.

In order to overcome all of these problems, an additional state variable should be introduced to the land slide model which represents the rate of change in cell height. In addition, some prediction rules (see Table 7.1) must be employed to let each cell predict its stopping points based on their heights and rates as well as the heights and rates of the neighboring cells. This rate-based quantized model is designed to be more accurate in representing landslide flows, but it requires more computing power. In addition, it introduces intelligent cells that send rate of flows to the neighbors instead of mass. Upon receiving these rates, cells calculate their overall rate of change and send it to all other neighbors. Then, each cell uses the prediction rules to calculate its nearest time advance. There are some neighboring cells that might have the same rate of change and they can not predict their next time step. In that case, they just stay on the phase of mass flow till they get a timed update from other cells so that they can update their heights according to their rate and the time they reach. Therefore, this last landslide model contains intelligent, predictive and self-updated cells that carry out the landslide flows more correctly than the previous models.

	$Rate_i > Rate_j$	$Rate_i < Rate_j$	
$\Delta z < 0$	$Z_i = Z_j + quantum$	$Z_i = Z_j$	$-\Delta z >= quantum$
$\Delta z = 0$	$Z_i = Z_j + quantum$	$Z_j = Z_i + quantum$	
$\Delta z > 0$	$Z_j = Z_i$	$Z_j = Z_i + quantum$	$\Delta z >= quantum$

Table 7.1: Prediction rules.

7.7 Models Development Experiences

The cellular model development environment allows the automated code generation. When developing a cellular DEVS model, the GUI is used to define the states variables, flow variables, variable types, ports, ports types, port to variable mapping, and cell's local transition function. In addition, the user can optionally enter the boundary conditions and any other helping function(s) that might be needed for the model.

Building the first model was done completely through the GUI except for the model parameters that were added to the beginning of the model code. In future work, the GUI can be easily extended to support this. Similarly, the other three models can be completely developed through the GUI and have their parameters added to the code. However, these models are using 3-D vectors in the flow calculations which are not yet supported as state variables in the environment. They can be either broken into three values that are supported by the environment and write the cell transition function accordingly or deal with them as objects that are added to the code of the generated model. The first option can be completely done through the GUI while it might degrade the simulation performance. On the other hand, the other option enhances the performance but requires the environment to be extended to support vectors in order to keep the code transparent to the user.

The last model required a special treatment of the flow variables that is not yet implemented in the environment. The environment assumes the flow variables should be reset after sending the flows into the neighboring cells which is the case in most of the flow based cellular models. The rate-based landslide model requires these flows not to be reset since each cell will need it in next iterations to update its state. Since, in this model, flow variables must be treated the same way as treating state variables, the generated code should be modified to account for this. Future improvement to the environment in this regard can be done by making the user select how the flow variables should be treated.

7.8 Experimental Results

The main purpose of the experiments was to show the speed up claimed to be achieved using the new approach and check the consistency of the practical measures to these theoretical claims. In addition to the landslide models, there are more models, not presented in this section, that show the significance of the approach presented in this dissertation (e.g. [11]). The landslide models were run using a 3.0 GHz Pentium 4 machine with 1GB of RAM. All runs were done for 32×32 cell space where the used data is an approximated portion taken from Fig.8 in [70]. The results presented here are execution times in seconds for 100 simulation iterations of all landslide models presented in the previous sections that were implemented using the new developed environment. Different runs with different setups were made for analyzing the different alternatives as well as comparing the new approach to the conventional cellular DEVS approach.

7.8.1 Modular vs. Non-Modular Approach

One of the main goals of this dissertation is to formulate and implement the idea of converting modular coupled cellular DEVS models into non-modular atomic ones in order to gain speedup in simulation. The comparison, in our landslide model context, was done by implementing the models using the conventional cellular DEVS approach and contrasting them with the new implementation that encapsulates the entire cell space in a non-modular atomic model. Table 7.2 shows the execution time in both approaches for all landslide models in seconds. The speedups, shown in the fourth column, were calculated by dividing the execution time of the conventional approach over the time for the new approach. The non-modular approach that is introduced in this work shows significant speedups in landslide models.

	Modular	Non-Modular	**Speedup**
Model-1	18	4	**4.5**
Model-2	110	15	**7.3**
Model-3	6	2	**3**
Model-4	10	1	**10**

Table 7.2: Execution and speedup of landslide models.

The speedup was gained from two major sources. The first one is the simulator-like enhancement in which the event list handler was optimized to efficiently manage the active cells. The other one is eliminating the inter-cell messages completely in the cell space model. Therefore, the speedup obtained reflects the percentage of average number of active cells per iteration to the total number of cells in the model ($M\times N$), and the total number of inter-cell messages generated during simulation. Both of these factors are application and model dependent which explains the differences in speedup between all models. The more inter-cell messages present, more is the speedup that can be achieved from the first source. On the other hand, the larger the percentage of active cells the lesser is the speed up gained from the other source.

$$speedup \propto \left(\frac{M \times N}{avg(activeCells)}, messages \right)$$

The first two landslide models represent discrete fixed step simulation approach in which all cells are active in all iterations. This means that the source of speedup in these two models came only from the inter-cell messages elimination. The second model gained more speed up than the first one because of the very small time step in differential equations that results in more landslide activities and hence more inter-cell messages. Speedup in the other two models is a result of both factors combined. The last model is the one that achieved the highest speed up among all models since it is heavily based on message passing because of the quantization and the rate-based prediction that results in very small time advances. In conclusion, the above speedup equation can be generalized to all cellular models using the new approach which model cell space applications as one atomic model.

7.8.2 Using the New Implementation for Conventional Cellular Models

Despite the significant speedup achieved in the non-modular implementation of the landslide models, the new approach introduced some forms of overhead and extra memory requirements to the atomic model. Table 7.3 illustrates the overhead introduced to the atomic cell space when running each block that employs the event list handler to simulate a single cell. This overhead resulted from the fact that, in each single iteration of an active block, the cell should be added to the list, extracted from the list, and then the computation takes place while in the conventional cellular DEVS implementation, cells do the computation without the list overhead. In addition, the new block implementation requires more storage memory since all cell states and variables are stored in arrays. This explains the missing result in Table 7.3 in which the machine ran out of memory to simulate that model which consists of 1024 blocks each having 18 (3×3) arrays. However, the full decomposition of cell space will result in a block that has 18 (34×34) arrays. The trade off between overhead, memory requirements, and speed up is in favor of the approach of this dissertation in case the cell space is fully decomposed into one atomic model.

	32×32 blocks	32×32 cells
Model-1	20	18
Model-2	114	110
Model-3	7	6
Model-4	-	10

Table 7.3: Comparing block and cell implementation of 32×32 Cell space.

7.8.3 Different Blocking Setting (Living with Messages)

In addition to the event list overhead, results show computational overhead in the approach in some block settings. This overhead associated with the blocking setups that divide the space into blocks that are required to communicate with each other via messages. Table 7.4 shows the execution time of different blocking setup starting with all cells inside one block (block size is 32×32) and ending with the setup in which each block contains one cell (block size is 1×1). All other setups, in between, result in execution time that is worse than the conventional cellular DEVS approach. That means that the blocking scheme endures a huge overhead when it starts communicating with other blocks.

Block size	32×32	16×16	8×8	4×4	2×2	1×1
Model-1	4	40	50	43	29	20
Model-2	15	970	767	440	221	114
Model-3	2	28	21	19	174	7
Model-4	1	24	17	15	13	-

Table 7.4: The new implementation with different blocking setups

This overhead inherited from the DEVS simulator implementation in which, when a block is about to receive an external message, the simulator informs it with the new message but it is required to iterate over the ports to decide which port is receiving the message. In the new approach, it will require iterating over all the boundary cells to get the message which results in (2W+2H-4) iterations in a W×H block to just receive a single message for one cell. On the other hand, conventional cellular DEVS implementation does the iteration once for the receiving cell. Therefore, the source of overhead is letting the block iterates over a big number of cells that might not receive external messages. The overhead factor can be represented by the following ratio (2W+2H+4)/X where X is the average cells that are receiving messages per iteration. In highly active cells where all boundary cells are receiving messages, the ratio will be one and all iterations will be worthwhile. In this case the overhead disappears compared to the conventional approach. The worst case can be represented by X<<(2W+2H+4) where the overhead factor is huge because of the extremely small number of receiving cells compared to the number of boundary cells which results in a huge number of unnecessary iterations. The solution to this problem is the simulator enhancement in which the messages should be encoded [88] in order to make the block know the receiving cell without going through all unnecessary iterations.

CHAPTER 8 : CONCLUSIONS AND FUTURE WORK

Conventional modular approaches of modeling cellular DEVS models were found to poorly perform in the case of very large scale spaces with high cell activities. Some related works were done to speedup simulations by the means of simulator enhancements that deal efficiently with the big volume of communication messages. Despite all of these enhancements, the models still spend a large amount of computational time in dealing with messages rather than spending all computational efforts in the actual model tasks.

In this dissertation, a new formalism was introduced to specify the cellular DEVS models in an efficient non-modular form. The new formalism was formulated using the closure under coupling property of DEVS in order to ensure equivalency of the models to their modular counterparts in parallel DEVS. Non-modular Models that were developed using the new cellular DEVS specification outperformed their modular equivalents. The speedup was gained from two sources. The first one is the efficient scanning of active cells which also can be achieved using simulator enhancements. The other one is the elimination of the inter-cell messages by fully decomposing the cellular space model into atomic one. However, specifying large and complex cellular models using the new specification was found to be complicated and difficult to verify. Therefore, different layer of formalism was introduced to allow simple and fast user specification of the efficient models. The new multi-layer formalism was made as generic as possible to support all cellular models that are currently supported by the parallel DEVS formalism like CA, PDE, and discrete time models.

The new formalism supports the automation of specification conversion between the different layers which made it possible to develop an automated environment that converts user specifications into cellular DEVS as well as parallel DEVS specifications. The new development environment supports model development through GUI where the user specification is input. It also supports automated conversion of user specification into the new cellular DEVS specification as well as automatic code generation that put the new form in parallel DEVS specification. This was done to allow possibility of running generated models in a standard parallel DEVS simulator. Future work might relax this constraint and introduce a special efficient cellular DEVS simulator.

The new environment was designed to make the model development faster and make the coding level transparent to the user. However, once the model code is generated, the user is granted full access to the code in order to modify the model and/or add more specific requirements that are not supported. The environment was found to support wide varieties of modeling requirements and it can be easily adapted to include more aspects in the future. It was tested using some of the standard software testing strategies and was found to perform well according to the test cases presented in chapter 6. The testing was extended to the models generated by this environment in order to verify that the environment generated what the user intended to develop. In addition to the software testing, the model is tested using the simulation-based DEVS verification approaches. A modification to the current automated cellular DEVS verification approach was proposed to correctly test models that account for zero time transitions. The last test presented in this work verified that the new approach is equivalent to the conventional cellular DEVS implementation through simulation methods in all models developed during this work. All automated testing classes are made available for the user to call in order to test and verify the developed models.

The fully decomposed cell spaces gave the best performance among all other blocking setups. The process of decomposing coupled model into atomic one involves adding simulator tasks into the new atomic model. These tasks include scanning active cells and handling the list of future events. The faster the event list handler, more is the speedup that can be gained in the decomposed model. Chapter 5 was dedicated to find the best event list handler for our environment which concluded that large decomposed cellular DEVS models prefer lists that have *O(1)* add and remove operations. That finding was based on the analytical as well as the experimental point of view. The analytical approach took place in the final format of the new implemented formalism in which the *add* and *remove* were found to be the most frequent operations in our design which was also supported by actual operation counts in some models. The selected implementation in the new environment is a standard unsorted array implementation. Since event lists might perform differently in different application, the event list class can be replaced, when needed, with another that should follow the design requirements listed in chapter 5.

The landslide models introduced in this dissertation tried to push the new environment to the limits. These complex models required more modifications to be added to the generated code. It was shown that all of the requirements can be easily added to the environment in future work and limit the need to modify the generated models. Experimental results showed that cellular models that have more inter-cell messages achieved more speedup when modeled in the new fully decomposed non-modular cellular DEVS specification. The other setups of blocking, where messages not entirely eliminated, showed significant overhead which resulted from the standard DEVS simulator implementation. The scope of this work does not include the distributed application of the new formalism in which a solution to this overhead will be a must. The solution will reside in the simulator enhancement in which message encoding scheme should be employed to include the receiving cell ID within all inter-cell messages.

Future work of this research might include the following:

- Modify the multi-component DEVS formalism in order to be equivalent to the parallel DEVS and compare it to the formalism introduced in this dissertation.
- Since this work targeted the modeling enhancement, a more integrated approach can be done by introducing an efficient simulation engine that is dedicated for the newly introduced formalism.
- The new development environment can be improved to include all the missing requirements, abstraction, and modifications that are needed by the complex models.
- A visualization environment of the cellular DEVS models can be designed and implemented to support animating the simulations.

- Extending the use of the new formalism to the distributed simulations which will include reducing the overhead that might be caused by the new implementation in order to utilize the advantage of the speedups achieved.

REFERENCES

[1] G. A. Wainer, "Modeling and simulation of complex systems with cell-DEVS," in *WSC '04: Proceedings of the 36th Conference on Winter Simulation,* 2004, pp. 49-60.

[2] G. A. Wainer and N. Giambiasi, "Cell-DEVS/GDEVS for Complex Continuous Systems," *Simulation,* vol. 81, pp. 137-151, 2005.

[3] A. Muzy and J. J. Nutaro, "Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators," in *1st Open International Conference on Modeling & Simulation (OICMS),* 2005.

[4] X. Hu and B. P. Zeigler, "A high performance simulation engine for large-scale cellular DEVS models," in *High Performance Computing Symposium (HPC'04), Advanced Simulation Technologies Conference,* 2004.

[5] G. Wainer and N. Giambiasi, "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation," *Simulation,* vol. 76, pp. 22-39, 2001.

[6] B. P. Zeigler, T. G. Kim and H. Praehofer, *Theory of Modeling and Simulation.* San Diego, CA, USA: Academic Press, Inc, 2000.

[7] E. Kofman and S. Junco, "Quantized-state systems: a DEVS Approach for continuous system simulation," *Trans. Soc. Comput. Simul. Int.,* vol. 18, pp. 123-132, 2001.

[8] T. Beltrame and F. E. Cellier, "Quantised state system simulation in Dymola/Modelica using the DEVS formalism," in *Proceedings 5th International Modelica Conference,* 2006, pp. 73-82.

[9] W. B. Lee and T. G. Kim, "Simulation speedup for DEVS models by composition-based compilation," in *Proceedings of Summer Computer Simulation Conference,* 2003, pp. 395-400.

[10] T. Beltrame, "Design and Development of a Dymola/Modelica Library for Discrete Event-oriented Systems Using DEVS Methodology," 2006.

[11] F. A. Shiginah and B. P. Zeigler, "Transforming DEVS to non-modular form for faster cellular space simulation," in *Proceedings of 2006 DEVS Symposium,* 2006, pp. 86-91.

[12] B. Chopard and M. Droz, *Cellular Automata Modeling of Physical Systems.* Cambridge University Press, 1998.

[13] T. Yu and S. Lee, "Evolving cellular automata to model fluid flow in porous media," in *EH '02: Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware (EH'02),* 2002, pp. 210.

[14] M. V. Avolio, G. M. Crisci, D. D'Ambrosio, S. Di-Gregorio, G. Iovine, R. Rongo and W. Spataro, "An extended notion of cellular automata for surface flows modeling," *WSEAS Transactions on Computers,* vol. 2, pp. 1080-1085, 2003.

[15] M. V. Avolio, G. M. Crisci, D. D'Ambrosio, S. Di-Gregorio, G. Iovine, V. Lupiano, R. Rongo and W. Spataro, "Surface flows modeling: Cellular automata simulation of lava, debris and pyroclastic flows," in *Proceedings of the iEMSs Third Biennial Meeting: "Summit on Environmental Modeling and Software",* 2006.

[16] B. D. Malamud and D. L. Turcotte, "Cellular-Automata Models Applied to Natural Hazards," *Computing in Science and Engineering,* vol. 2, pp. 42-51, 2000.

[17] N. Ganguly, P. Maji, S. Dhar, B. K. Sikdar and P. P. Chaudhuri, "Evolving cellular automata as pattern classifier," in *ACRI '01: Proceedings of the 5th International Conference on Cellular Automata for Research and Industry,* 2002, pp. 56-68.

[18] P. Maji, C. Shaw, N. Ganguly, B. K. Sikdar and P. P. Chaudhuri, "Theory and application of cellular automata for pattern classification," *Fundam. Inf.,* vol. 58, pp. 321-354, 2003.

[19] T. N. Mudge, R. A. Rutenbar, R. M. Lougheed and D. E. Atkins, "Cellular image processing techniques for VLSI circuit layout validation and routing," in *DAC '82: Proceedings of the 19th Conference on Design Automation,* 1982, pp. 537-543.

[20] X. Yang, "Characterization of multispecies living ecosystems with cellular automata," in *ICAL 2003: Proceedings of the Eighth International Conference on Artificial Life,* 2003, pp. 138-141.

[21] A. Dupuis and B. Chopard, "Parallel simulation of traffic in Geneva using cellular automata," pp. 89-107, 2001.

[22] G. Wainer, "ATLAS: A language to specify traffic models using Cell-DEVS," *Simulation Modeling Practice and Theory,* vol. 14, pp. 313-337, 2006.

[23] R. O. Cunha, A. P. Silva, A. A. F. Loureiro and L. B. Ruiz, "Simulating large wireless sensor networks using cellular automata," in *ANSS '05: Proceedings of the 38th Annual Symposium on Simulation,* 2005, pp. 323-330.

[24] R. Hu and X. Ruan, "Differential equation and cellular automata model," in *IEEE International Conference on Robotics, Intelligent Systems and Signal Processing,* 2003, pp. 1047-1051.

[25] M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach.* Secaucus, NJ, USA: Springer-Verlag New York, Inc, 2001.

[26] G. Spezzano and D. Talia, "A high-level cellular programming model for massively parallel," in *2nd Int. Workshop on High-Level Programming Models and Supportive Environments (HIPS97),* 1997.

[27] D. Talia, "Cellular Processing Tools for High-Performance Simulation," *Computer,* vol. 33, pp. 44-52, 2000.

[28] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling.* Cambridge, MA, USA: MIT Press, 1987.

[29] S. Wolfram, *A New Kind of Science.* Champaign, Illinois, US, United States: Wolfram Media Inc, 2002.

[30] J. Ameghino, A. Troccoli and G. Wainer, "Models of complex physical systems using cell-DEVS," in *SS '01: Proceedings of the 34th Annual Simulation Symposium (SS01),* 2001, pp. 266.

[31] A. Muzy, E. Innocenti, A. Aiello, J. -. Santucci and G. Wainer, "Cell-DEVS quantization techniques in a fire spreading application," in *WSC '02: Proceedings of the 2002 Winter Simulation Conference (WSC'02) - Volume 1,* 2002, pp. 542-549.

[32] G. Wainer and R. Madhoun, "Creating Spatially-Shaped Defense Models Using DEVS and Cell-DEVS," *JDMS: The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology,* vol. 2, pp. 121-143, 2005.

[33] A. C. Chow, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism and its distributed simulator," *Trans. Soc. Comput. Simul. Int.,* vol. 13, pp. 55-67, 1996.

[34] C. Chow and B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism," in *WSC '94: Proceedings of the 26th Conference on Winter Simulation,* 1994, pp. 716-722.

[35] G. A. Wainer and N. Giambiasi, "N-dimensional Cell-DEVS Models,"*Discrete Event Dynamic Systems,* vol. 12, pp. 135-157, 2002.

[36] G. Wainer and N. Giambiasi, "Timed cell-DEVS: modeling and simulation of cell spaces," pp. 187-214, 2001.

[37] G. Wainer, "Performance analysis of continuous cell-DEVS models," in *Proceedings of 18th European Simulation Multiconference,* 2004.

[38] H. Saadawi and G. Wainer, "Modeling a sand pile application using cell-DEVS," in *Proceedings of the 2003 SCS Summer Computer Simulation Conference,* 2003.

[39] G. A. Wainer, "Modeling and simulation of complex systems with cell-DEVS," in *WSC '04: Proceedings of the 36th Conference on Winter Simulation,* 2004, pp. 49-60.

[40] F. P. Wyman, "Improved event-scanning mechanisms for discrete event simulation," *Commun. ACM,* vol. 18, pp. 350-353, 1975.

[41] J. G. Vaucher and P. Duval, "A comparison of simulation event list algorithms," *Commun. ACM,* vol. 18, pp. 223-230, 1975.

[42] J. S. Steinman, "Discrete-event simulation and the event horizon part 2: Event list management," in *PADS '96: Proceedings of the Tenth Workshop on Parallel and Distributed Simulation,* 1996, pp. 170-178.

[43] R. Rönngren, J. Riboe and R. Ayani, "Lazy queue: An efficient implementation of the pending-event set," in *ANSS '91: Proceedings of the 24th Annual Symposium on Simulation,* 1991, pp. 194-204.

[44] W. M. McCormack and R. G. Sargent, "Analysis of future event set algorithms for discrete event simulation," *Commun. ACM,* vol. 24, pp. 801-812, 1981.

[45] D. W. Jones, "An empirical comparison of priority-queue and event-set implementations," *Commun. ACM,* vol. 29, pp. 300-311, 1986.

[46] K. Chung, J. Sang and V. Rego, "A performance comparison of event calendar algorithms: an empirical approach," *Softw. Pract. Exper.,* vol. 23, pp. 1107-1138, 1993.

[47] H. Bahr and R. DeMara, "Smart priority queue algorithms for self-optimizing event storage," *Simulation Modeling Practice and Theory,* vol. 12, pp. 15-40, April. 2004.

[48] M. A. Weiss, *Data Structures and Algorithm Analysis in Java.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 1998.

[49] R. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in Java.* John Wiley & Sons, Inc, 2000.

[50] G. M. Adelson-Velskii and E. M. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady,* vol. 3, pp. 1259-1262, 1962.

[51] J. Baer and B. Schwab, "A comparison of tree-balancing algorithms," *Commun. ACM,* vol. 20, pp. 322-330, 1977.

[52] Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer,* vol. 34, pp. 135-137, 2001.

[53] Burnstein, *Practical Software Testing.* New York, USA: Springer-Verlag New York, Inc, 2003.

[54] L. Copeland, *A Practitioner's Guide to Software Test Design.* Norwood, MA, USA: Artech House, Inc, 2003.

[55] M. Ellims, J. Bridges and D. C. Ince, "The Economics of Unit Testing," *Empirical Softw. Engg.,* vol. 11, pp. 5-31, 2006.

[56] N. Juristo, A. M. Moreno and S. Vegas, "Reviewing 25 Years of Testing Technique Experiments,"*Empirical Softw. Engg.,* vol. 9, pp. 7-44, 2004.

[57] A. Whittaker, "What Is Software Testing? And Why Is It So Hard?" *IEEE Software,* vol. 17, pp. 70-79, 2000.

[58] B. P. Zeigler, *Objects and Systems: Principled Design with Implementations in C++ and Java.* New York, NY, USA: Springer-Verlag New York, Inc, 1997.

[59] M. Pol, R. Teunissen and E. V. Veenendaal, *Software Testing: A Guide to the TMap Approach.* Boston, MA, USA: Addison-Wesley, 2001.

[60] O. Balci, "Principles and techniques of simulation validation, verification, and testing," in *WSC '95: Proceedings of the 27th Conference on Winter Simulation,* 1995, pp. 147-154.

[61] R. G. Sargent, "Validation and verification of simulation models," in *WSC '04: Proceedings of the 36th Conference on Winter Simulation,* 2004, pp. 17-28.

[62] Y. Labiche and G. Wainer, "Towards the verification and validation of DEVS models," in *Proceedings of 1st Open International Conference on Modeling & Simulation,* 2005, pp. 295-305.

[63] M. H. Hwang, "Tutorial: Verification of real-time system based on schedule-preserved DEVS," in *Proceedings of 2005 DEVS Symposium,* 2005.

[64] M. H. Hwang and B. P. Zeigler, "A modular verification framework using finite and deterministic DEVS," in *Proceedings of 2006 DEVS Symposium,* 2006, pp. 57-65.

[65] B. P. Zeigler, "Verification and validation of DEVS models: Applying the theory of modeling and simulation to the needs of simulation based acquisition," in *Proceedings of Summer Computer Simulation Conference,* 2000.

[66] G. Wainer, L. Morihama and V. Passuello, "Automatic verification of DEVS models," in *Proceedings of the 2002 Spring Simulation Interoperability Workshop,* 2002.

[67] N. J. Nilsson, *Artificial Intelligence: A New Synthesis.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 1998.

[68] N. Dawson, Q. Du and T. F. Dupont, "A finite difference domain decomposition algorithm for numerical solution of the heat equation," *Mathematics of Computation,* vol. 57, pp. 63-71, July. 1991.

[69] H. Chen and C. F. Lee, "Numerical simulation of debris flows," *Canadian Geotechnical Journal,* vol. 37, pp. 146-160, 2000.

[70] D'Ambrosio, S. D. Gregorio and G. Iovine, "Simulating debris flows through a hexagonal cellular automata model: SCIDDICA S_3-hex," *Natural Hazards and Earth System Sciences,* vol. 3, pp. 545-559, 2003.

[71] Dattilo and G. Spezzano, "Simulation of a cellular landslide model with CAMELOT on high performance computers," *Parallel Comput.,* vol. 29, pp. 1403-1418, 2003.

[72] P. D'Odorico and S. Fagherazzi., "A probabilistic model of rainfall-triggered shallow landslides in hollows: A long-term analysis," *Water Resour. Res.,* vol. 39, 2003.

[73] Gascuel, M. Cani, M. Desbrun, E. Leroy and C. Mirgon, "Simulating landslides for natural disaster prevention," in *Eurographics Workshop on Computer Animation and Simulation (EGCAS),* 1998.

[74] S. Hergarten, "Landslides, sandpiles, and self-organized criticality," *Natural Hazards and Earth System Sciences,* vol. 3, pp. 505-514, 2003.

[75] O. Hungr, "Flow slides and flows involving granular soils," in *International Workshop on: Occurrence and Mechanisms of Flows in Natural Slopes and Earthfills,* 2003.

[76] J. Lin and C. Ku, "Simulation of slope failure using a meshed based partition of unity method," in *15th ASCE Engineering Mechanics Conference,* 2002.

[77] E. Segre and C. Deangeli, "Cellular automaton for realistic modeling of landslides," *Nonlinear Processes in Geophysics,* vol. 2, pp. 1-15, 1995.

[78] N. Sitar and M. M. MacLaughlin, "Kinematics and discontinuous deformation analysis of landslide movement," in *2nd Panamerican Symposium on Landslides,* 1997.

[79] Yvonne and C. Michael, "Numerical modeling of landscape evolution: geomorphological perspectives," *Prog. Phys. Geogr.,* vol. 28, pp. 317-339, 2004.

[80] K. Christensen and Z. Olami, "Scaling, phase transitions, and nonuniversality in a self-organized critical cellular-automaton model," *Physical Review A (Atomic, Molecular, and Optical Physics),* vol. 46, pp. 1829-1838, Aug. 1992.

[81] D. A. Head and G. J. Rodgers, "Crossover to self-organized criticality in an inertial sandpile model," *Phys Rev E.,* vol. 55, pp. 2573-2579, 1997.

[82] S. Hergarten and H. J. Neugebauer, "Self-organized criticality in a landslide model," *Geophys. Res. Lett.,* vol. 25, pp. 801-804, 1998.

[83] P. Bak, *How Nature Works: The Science of Self Organized Criticality.* New York, NY, USA: Springer-Verlag New York, Inc, 1996.

[84] C. R. Calidonna, C. D. Napoli, M. Giordano, M. M. Furnari and S. D. Gregorio, "A network of cellular automata for a landslide simulation," in *ICS '01: Proceedings of the 15th International Conference on Supercomputing,* 2001, pp. 419-426.

[85] J. Ohta and I. Matsuba, "Analysis of earthquakes based on a dissipative cellular-automata model," *Electronics and Communications in Japan, Part 3,* vol. 82, pp. 20-27, 1999.

[86] P. G. Akishin, M. V. Altaisky, I. Antoniou, A. D. Budnik and V. V. Ivanov, "Simulation of earthquakes with cellular automata," *Discrete Dynamics in Nature and Society,* vol. 2, pp. 267-279, 1998.

[87] A. Muzy, E. Innocenti, J. Santucci and D. R. C. Hill, "Optimization of cell spaces simulation for the modeling of fire spreading," in *ANSS '03: Proceedings of the 36th Annual Symposium on Simulation,* 2003, pp. 289.

[88] J. S. Lee, "Space-based data management for high performance distributed simulation," 2001.