Development of a flexible parameterization and a dynamic batch processing for the AMIRIS model

Nibir Hossain

13 January, 2014

Contents

Acknowledgements

First of all, I would like to offer my profound gratitude and earnest thanks to Prof. Göhner for giving me the opportunity to conduct the thesis work and for providing all kind of support during my thesis work.

I am really grateful to Dr. Nasser Jazdi for his supervision. I want to give my sincere thanks to him for his support and helpful advice during my thesis.

I would like to give my special thanks to Dr. Marc Deissenroth and Dipl. Ing. Matthias Reeg for paving the proper direction to complete my thesis work. I want to provide my heartiest gratitude to Dr. Marc Deissenroth for his acceptance, support, supervision, useful advice and friendly relation during my thesis work.

I would also like to thank all the colleagues at Department of Systems Analysis and Technology Assessment, Institute of Technical Thermodynamics, German Aerospace Center for their valuable help and advice especially to Dr. Christoph Schillings, Dr. Dominik Heide and Dr. Uwe Pfenning.

Last not least, I want to provide heartiest love to my family and my beloved grandmother for supporting me at every step of my life. Moreover, I would like to give thanks to all my friends for being with me.

Sincerely yours

Nibir Hossain

Abbreviations

AMIRIS **A**gent-based **M**odel for the **I**ntegration of **R**enewables **I**nto the Power **S**ystem

MAS **M**ulti-**A**gent **S**ystem

ECO **E**lectricity **C**ommunication **O**bject

IT **I**nformation **T**echnology

ABMS **A**gent-**b**ased **M**odeling and **S**imulation

ABM **A**gent-**b**ased **M**odel

DAI **D**istributed **A**rtificial **I**ntelligence

DBMS **D**ata**b**ase **M**anagement **S**ystem

OS **O**perating **S**ystem

AI **A**rtificial **I**ntelligence

BDI **B**elief **D**esire **I**ntention

KQML **K**nowledge **Q**uery **M**anipulation **L**anguage

ACL **A**gent **C**ommunication **L**anguage

FIPA **F**oundation for **I**ntelligent **P**hysical **A**gents

RES	**R**enewable **E**nergy **S**ources
FIT	**F**eed-**i**n-**T**ariffs
TSO	**T**ransmission **S**ystem **O**perator
PPO	**P**ower **P**lant **O**perator
CSV	**C**omma **S**eparated **V**alues
XML	**Ex**tensible **M**arkup **L**anguage
RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
OLAP	**O**nline **A**nalytical **P**rocessing
XSD	**X**ML **S**chema **D**efinition
DTD	**D**ocument **T**ype **D**efinition
ORM	**O**bject **R**elational **M**odel
API	**A**plication **P**rogramming **I**nterface
DOM	**D**ocument **O**bject **M**odel
SAX	**S**imple **A**PI for **X**ML
StAX	**St**reaming **A**PI for **X**ML
JAXB	**J**ava **A**rchitecture for **X**ML **B**inding
POJO	**P**lain **O**ld **J**ava **O**bject
CDB	**C**apacity **D**ata**b**ase

GUI **G**raphical **U**ser **I**nterface

IDE **I**ntegrated **D**evelopment **E**nvironment

Abstract

Multi-agent system (MAS) is a popular and prominent subject in computing world from the last three decades. MAS is used in many important applications and domains such as market analysis, business process management, distributed systems and information management systems. AMIRIS model is a multiagent modeling and simulation approach which is developed for analyzing the integration of renewables into the electricity markets by the direct marketing.

This thesis is divided into two main subcategories: *dynamic and flexible parameterization of the model criteria and of different agents* as well as *batch processing* of the AMIRIS program. The agents in AMIRIS model require considerable amount of parameters for instantiating themselves, communicating and coordinating between them. The AMIRIS model also needs necessary parameters to define the criteria of the model. A dynamic and flexible parameterization mechanism has been developed to automatically adapt the parameters for instantiating the agents and specifying the model criteria in the runtime. On the other hand, the AMIRIS model needs the recursive execution with different set of parameters for producing different set of outputs for better statistical analysis. External Java batch processing application has been established to manage multiple individual executions of the AMIRIS model in a single run. In order to develop parameterization and batch processing of the AMIRIS model, different parameterization mechanisms and batch processing methodologies have been evaluated and the most suitable for the AMIRIS model has been chosen. The successful implementation of parameterization and batch processing is shown by means of example runs.

By the development of the approaches the execution time of the model is reduced approximately by 32%, inclusion of a new agent via parameterization to the model is simplified, and the user interaction to operate the model is significantly reduced.

Chapter 1

Introduction

1.1 Context and Problem

Multiagent systems are significantly used in distributed applications, artificial intelligence and computer science, for example, market analysis, business process management, distributed sensing and human-computer interfaces. AMIRIS is a multiagent simulation model which is used to analyze the direct marketing and market integration of the renewable energy. The focus of the communication among agents in the AMIRIS model is the *electricity communication object* (ECO) that contains all the relevant information about the electricity trade and the involved actors (agents in the model). ECO can be interpreted as *an electricity commercial contract* between two parties. Previously the model was designed with static ECOs between agents and the parameters of ECOs were written in the source code. The agents were not able to change their electricity contracts with their necessity in the runtime. For producing different output for different runs, developers were required to change manually the parameters of the ECOs of the agents before every run. Moreover, the inclusion of new agents to the model was also difficult. In order to add a new agent, the developers were required to write a considerable amount of lines of code which is time consuming and cost ineffective. In addition, the frequent changing of the source code is error-prone.

On the other hand, most of the simulation models need iterative execution with different parameters for better statistical analysis of the generated outputs. But the AMIRIS model was able to execute once in a run. For every individual run of the simulation, the user interacted with the graphical user interface to initialize the model criteria and the agents which is time consuming, and thus frequent user interaction made profound statistical analysis impossible.

The plant operators are differentiated per technology generation (i.e. wind, solar and biomass) up to 28 types (per 4 remuneration technology classes and up to 7 types of ownership). There are a total of up to 10 different types of intermediaries. So there is a maximum number of ECO up to 840 contracts. All of the contracts are defined in the source code in static way which are not possible to change in the runtime.

The problems in the AMIRIS model are defined specifically in the following.

- Cumbersome and difficult to grasp the parameters of the agents and especially the *electricity communication object*.
- Documentation of initial settings and the simulation input parameters is complicated.
- Multi-simulations of various scenarios with modified parameters of the agents are not automatically possible (batch processing).

The above aspects are being challenged by developing *a dynamic and flexible parameterization mechanism* as well as *batch processing* for the AMIRIS model. The parameterization mechanism provides the flexibility to alter the parameters of the model criteria and the agents with their requirements in the runtime. On the other hand, batch processing provides the facility to the AMIRIS model for multiple individual executions of the model in a single run. The development of these mechanisms reduces the effort of the developers and the interaction of the users significantly.

1.2 Goal and Objectives

In the thesis work, one of the main goals is the development of a dynamic and flexible parameterization for initializing the agents and for defining the criteria of the AMIRIS model. Another one is the implementation of a suitable batch processing for recursive execution of the model in a single run.

In the dynamic and flexible parameterization approach, first of all, different storage applications have to be analyzed and evaluated for selecting an appropriate application to store the parameters for defining the model criteria and for initializing the agents. The validation mechanisms will have to be then explored for validating the parameters. After that the parser mechanisms will be examined and assessed to choose a suitable and efficient parser for parsing the parameters from the storage application to define the model criteria and instantiate the agents as well as setup the electricity communication objects between agents in the AMIRIS model. Finally a dynamic and flexible approach for the parameterization has to be developed.

For the development of batch processing for the AMIRIS model, several batch processing mechanisms have to be explored, examined and a suitable one has to be developed for managing the multiple individual executions of the model in a single run.

After developing both approaches, the AMIRIS model will have to be tested. Finally the performance and improvement of the AMIRIS model have to be analyzed and results will be shown.

1.3 Thesis Outline

After the introduction of this thesis work, the second chapter introduces basics on software agent and multiagent systems. Firstly, it defines a software agent, and describes different environments of the agents, characteristics of the agents and different architectures of the agents. Furthermore, this chapter discusses different characteristics of multiagent systems as well as communication and coordination between different agents in multiagent systems. Finally, this chapter gives an overview of agent-based modeling and simulation.

An overview of the AMIRIS model is demonstrated in the third chapter. This chapter firstly introduces the policy framework of the market integration of renewable energy. Then different actors who are involved in the energy market are discussed. Moreover, it describes the structure of the AMIRIS model.

The fourth chapter describes and analyzes parameterization mechanism for the parameters of the AMIRIS model and batch processing for managing multiple individual executions of the AMIRIS model in a single run. First of all, it discusses and assesses different data storage applications to store and manage the parameters for defining the model criteria and for initializing different agents of the AMIRIS model. This chapter then describes and evaluates different XML validation mechanisms for validating the parameters. After that it depicts and valuate different parser mechanisms to process the parameters from the selected data storage application. Finally this chapter analyzes some batch processing mechanisms for recursive execution of the AMIRIS model in a single run.

The developed prototype and implementation of parameterization and batch processing are depicted in the fifth chapter. At first, this chapter describes the system architecture with different components. Then it demonstrates the implementation of dynamic and flexible parameterization mechanism for processing the parameters of the AMIRIS model. After that it illustrates the development of batch processing mechanism for the AMIRIS model. At last this chapter summaries the results and evaluates the performance of the

AMIRIS model with sensitivity analysis.

Finally, the sixth chapter concludes this thesis work with a short description of future work.

Chapter 2

Basics on Agents and Multiagent Systems

The complexities of distributed computing and the limitation of human-to-program interfaces are vital issues in Information Technology (IT). The programmer predicts, plans and codes explicitly for every action of the computer program. If the design of the program is not well-anticipated, the system can crash which might be cause of serious disasters such as plane crash and malfunctioning of medical instruments. In order to overcome these obstacles, such computer systems are required which can make the *decision for themselves* what they need to do for achieving their desired goals. These systems are called *agents*. An agent which can perform its activities robustly in a changing, dynamic, unpredictable and open environment, where there is remarkable possibility to fail the actions, is called *intelligent agent*. Section 2.1 introduces the term *agent*, and different *properties of environment, characteristics and architectures* of the agents.

A multi-agent system (MAS) is a set of interactive, proactive agents that communicate and interact with each other in a common environment. It is applied to such problems which are very difficult to resolve by a single agent or a monolithic system. The *characteristics* of multiagent systems as well as *agent communication and coordination* are discussed in section 2.2.

Agent-based modeling and simulation (ABMS) is a recent technique to analyze and model the dynamic and complex adaptive multiagent systems with autonomous, self-learning and interacting agents. Agent-based modeling (ABM) is used in many disciplines such as supply chains, consumer market analysis, military planning and economics. ABM approach has been used to develop the AMIRIS model. Section 2.3 represents an overview of ABMS.

2.1 Agents

Agents or software agents are computer systems which are able to perform the actions in the dynamic, unpredictable and open environment. A software agent is a computer system situated in an environment that acts on behalf of its user and is characterized by a number of properties [Chi03]. Agents have two important capabilities: *autonomous action* and *interaction* with other agents [Woo02].

Autonomous Action: The agents have the ability to take the decision for themselves for achieving their desired goals.

Interaction Capability: They have the capability to communicate with other agents for acquiring their design objectives.

This section gives a brief introduction of *intelligent agent*, describes different *environments, characteristics and architectures* of the agent.

2.1.1 Intelligent Agents

Software agents are used in different applications and domains of Information Technology. They are very popular and convenient in distributed artificial intelligence (DAI), database management systems (DBMS), operating systems (OS). Since about 1980, multiagent systems have been introduced and since about 1990s they have gained widespread recognition in the field of computer science. Multiagent systems are used in many important applications and domains such as market analysis, business process management, distributed systems and information management systems.

There is no specific definition for *an agent* and there are lot of debates and controversies about the definition of the term *agent*. Two most suitable and adopted definitions of an agent are presented below.

According to Dr. Danny B. Lange (1998): *An agent is a program that assists people and acts on their behalf. Agents function by allowing people to delegate work to them* [LaM98].

Albeit it is an accurate definition of an agent, it does not fulfil all of the purposes of an agent. Since an agent can be used in different areas such as operating systems, databases and mulitagent systems, it has different roles for different applications.

The definition of an agent is adapted from Wooldridge and Jennings (1995).

An agent is a computer system that is situated in some environment, and that is capable

of autonomous action in this environment in order to meet its design objectives [Woo02].

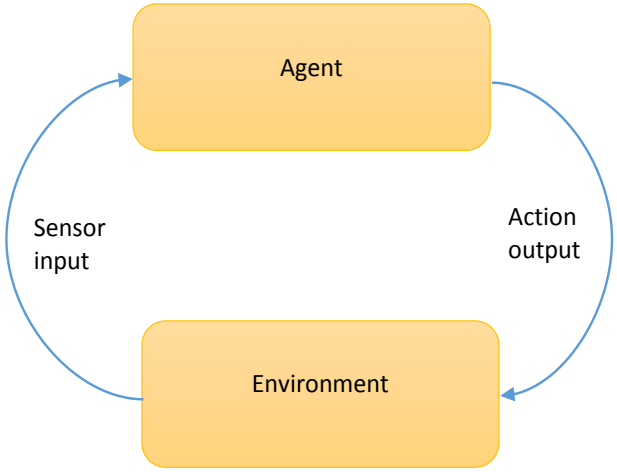

Figure 2.1: An agent in its environment. The agent takes the sensory input from the environment and produces as output actions that affect it. The interaction is usually an ongoing non-terminating one.

Source: Michael Wooldridge, (2002)

Figure 2.1 represents the abstract view of an agent. The agent generates *actions* to have influence on its environment. The agent does not have the *complete control* over its environment. It can *partially* control its environment. On the other hand, the agent observes the environment and takes the input from the environment to produce the expected output. The agent may perform an action twice or more in case of failure action.

2.1.2 Environments of Agents

An environment is a place where agents perform their actions for reaching their design objectives. The agents and environments are highly interrelated with each other. According to Russell and Norvig, the classification of environment properties is presented below [Rus95].

Fully Observable vs. **Partially Observable**: In order to perform the actions and take the optimal decisions, the agent needs to understand every state of its environment. If the agent can sense every point in time of the environment, this environment is called *fully observable*, otherwise it is called *partially or unobservable environment*. A *chess playing program* and *bridge playing system* are the examples in fully and partially observable environments respectively.

Deterministic vs. **Stochastic**: In the deterministic environments, the agent can perceive the next state of its environment using the current state and the agent's actions. An example of the application in deterministic environment is image processing. On the other hand, the next state can be totally unknown to the agent in stochastic (non-deterministic) environment. *Taxi driving* is an example in a stochastic environment because of unpredicted behavior of the traffic. Moreover, the tires can blow out or the engine can stop without any warning.

Episodic vs. **Sequential**: The current action of an agent does not depend on the previous actions in episodic environment. In contrast, the actions are interconnected with the previous series of actions in sequential environment.

Static vs. **Dynamic**: The dynamic environment is changed with the actions and decision by the agent, whereas the static environment does not change its state.

Discrete vs. **Continuous**: In discrete environment, all actions are fixed and known to the agent. By contrast, the number of actions are infinite and the action can be unknown to the agent in continuous environment.

2.1.3 Characteristics of Agents

A software agent can have different characteristics in the multiagent systems. According to Wooldridge and Jennings, the agent possesses the following properties: *autonomy, reactivity, social ability, pro-activeness* and *flexibility*. Moreover, *cooperation* among different software agents may be very useful in achieving the objectives an agent has [Chi03]. Some important characteristics of an agent are discussed in the following.

Autonomy: The agent should performs its own tasks without direct interference of human or other agents. Though it cannot control its environment completely, it should have control on its own actions and influence on its environment.

Reactivity: The agent receives the input from its environment and processes the inputs to generate the expected output. It accomplishes some actions which can affect to change its environment.

Social Ability: The agent should interact with other agents and cooperate other agents to establish a suitable social environment. This means the agent should be affable and companionable.

Coordination: It means that the agent is able to perform some activity in a shared environment with other agents [Ode00]. Activities are often coordinated via plans, workflows,

or some other process management mechanism [Ode00].

Cooperation: This means that the agent may perform some actions with the coordination of other agents to achieve the common goals.

Flexibility: The system should be responsive, social and proactive so that the agent can understand and cope up with its environment [Jen98].

Learning: The agent should be capable of

- reacting flexibly to changes in its environment;
- taking goal-directed initiative, when appropriate; and
- learning from its own experience, its environment, and interactions with others [Chi03] [Syc98].

Mobility: The agent should have the capability to transport itself from one machine to another. It should be compatible with heterogeneous systems and platforms [Etz95].

Pro-activeness: It is not sufficient that the agent will perform some specific predefined actions. The agent should take some decisions and initiatives in case of crucial situation to achieve its objectives [Chi03] [Jen98] [Ode00].

Prediction Ability: The agent should be predictive so that it can perceive the future actions to produce the accurate output [Goo93].

2.1.4 Architectures of Agents

The architecture of the agent is the important process which defines the agent's behavior to adapt, perform actions and take decisions in dynamic, changing and open environment. The architecture development of the intelligent agent is the main concern in the area of agent-based modeling and simulation systems. The lower range from purely reactive or behavioral architectures based on *subsumption architecture* of Brooks [Broo91]. To more deliberative architectures that reason about their actions based on *belief, desire* and *intention* (BDI) model [RaGe95]. Different architectures are applied to different agents according to their requirements. The agent architectures have been categorized into four groups: *logic-based, reactive, belief-desire-intention* and *hybrid* architectures [Woo02].

Logic-based Architecture: Logic-based architecture is a *traditional* knowledge-based approach to construct an intelligent system with artificial intelligence (AI). It is also known as *symbolic* AI. This system can generate the intelligent behavior using a symbolic representation of its environment and the symbolic representations can be manipulated

syntactically. The symbolic representations are *logical formulae* and the syntactical manipulation are known as *logical deduction*. Since the human knowledge is symbolic, the encoding is simpler in logic-based architecture. For example, someone says "Adam loves Eve", he/she understands that 'Adam' is a symbolic representation of a person who loves 'Eve'. He/she also understands that 'loves' is a symbolic representation of an emotion between two people. The person can imagine that 'Adam' is perhaps male and 'Eve' is perhaps female. Different people can think in different ways about it. Because of complexity of symbolic relationships, it is very difficult to implement computationally.

Human can easily realize the logic because of computational building of the agent architecture. On the other hand, it is not easy to translate the real world. A considerable amount of time is required to accomplish the symbolic representation and manipulation.

Reactive architecture: The main problem in *logic-based (symbolic) architecture* is the *interaction* between agent and its environment. The logical representation is not sufficient to construct an agent that can interact with and perform its actions in its dynamic environment. Rodney Brooks has investigated alternative approach for agent architecture. He has included some new criteria and discarded some into/from symbolic AI paradigm. The new paradigm is called *reactive* architecture. The main focuses of reactive architecture are:

- The symbolic representations are discarded.
- The intelligent and rational behavior of an agent are highly interrelated with environment.
- The agent is able to emerge the intelligent behavior from a set of simpler behaviors.

The best-known reactive agent architecture is *subsumption architecture* which is described briefly in the following. The *subsumption architecture* is developed by Rodney Brooks. It has two important features: *decision-making of an agent* and *selection of suitable action*. The first characteristic of subsumption architecture is that the agent can make a decision by analyzing a set of *task accomplishing behaviors*. For this reason, reactive architecture is referred as *behavioral architecture*. Each behavior is considered as an individual *action method*. This method receives the input from the environment and maps the input to an action to perform. Every behavioral module is designed to achieve a particular goal. Brooks has implemented every module with a finite state machine so that every action can reach a specific goal.

Secondly, multiple behaviors can fire at a time in reactive architecture. An agent can choose the possible best action among different selected actions from the *action* methods.

Finally, by performing the appropriate action, the agent reaches its desired objective.

Belief-Desire-Intention (BDI) Architecture: It is a popular and well-known architecture for building the agent. The philosophical tradition of understanding *practical reasoning* is the root of BDI architecture. BDI architecture suggests the logical theory which defines the mental attitudes of belief, desire and intention of an agent. *Practical reasoning* is a combination of two important approaches. The first approach is that deciding *what objectives* an agent desires to obtain. Another one is that *how* the agent achieves these objectives.

Practical reasoning is a suitable example for better understanding of BDI paradigm. Suppose, a student receives his/her first degree with a good result. He/she has the two possibilities either the student can *become an academic* (if the result is not good, this option is not available) or can *enter the company*. When these options are available, the student has to *choose between them* and make a *commit* to a suitable one. The chosen alternatives are the *intentions*. The *intentions* help the agents to perform their appropriate actions to achieve their design objectives.

Layered (Hybrid) Architecture: This architecture is the combination of *logic-based* and *reactive* architectures. It creates separate subsystems which are used to conduct the different types of behaviors. The subsystems maintain a hierarchical model of interacting *layers*. All layers in layered architecture are interconnected with their immediate layers. The control flows and information are the crucial parts of the layered architecture. The control flow can be classified into two categories: *horizontal* and *vertical layering*.

Horizontal layering: Every layer is deliberately connected to the sensor input and action output in the horizontally layered architectures (Figure 2.2 (a)). The layers generate actions and suggest what actions should perform for achieving better outcomes.

Vertical layering: In the vertically layered architecture, all layers do not deal the sensor input and action output. Only two layers are responsible to deal the sensory input and the action output respectively. The vertical layered architecture is classified into *one-pass control* and *two-pass control architecture* (Figure 2.2 (b), (c)).

The main advantage of horizontally layered architecture is the simplicity of design of the architecture. Every agent can interact with n layers. Every layer consists of different actions and behavioral methods for individual action. The agents explore every layer to select the best possible actions for achieving their desired goals. During the interaction with different layers, the agents can also generate action suggestions in each layer. If the generated and existing behaviors are not consistent, the overall behavior of the agent will not be consistent. Another problem is, for example, a system has n number of layers and

each layer has m number of actions. If each agent interacts with n layers which have m actions individually, the number of interactions is m^n which can cause a bottleneck performance of the system. This means the performance of the system will be significantly reduced with the increasing number of layers and actions. A *mediator* method is used to make sure the consistency of the horizontally layered architecture. It maintains the control tracks to manage which agent interacts with which layer and in which time.

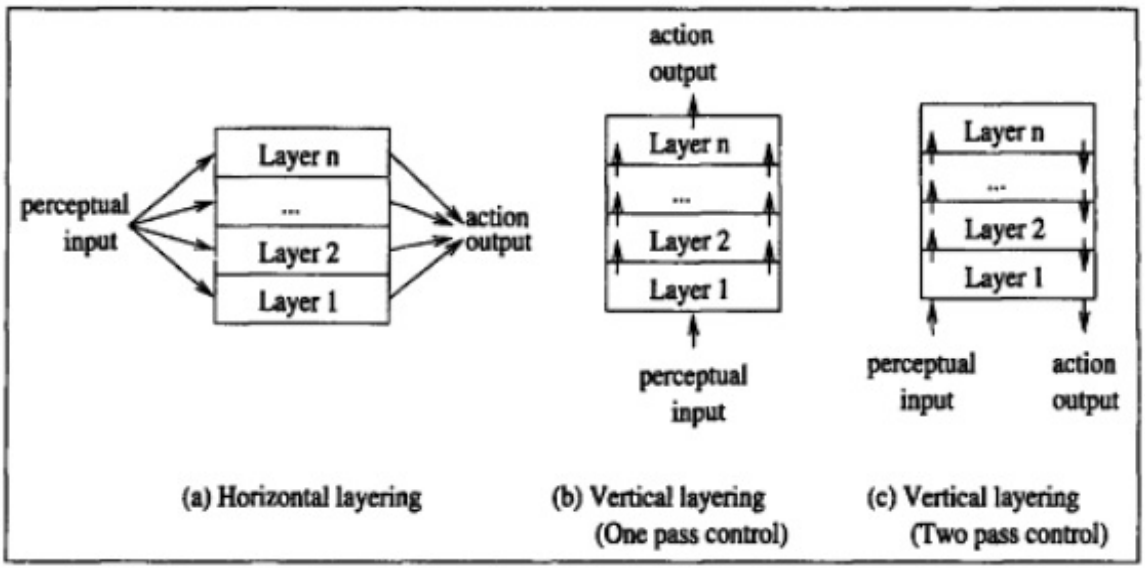

Figure 2.2: Information and control flows in three types of layered agent architecture.

Source: Michael Wooldridge, (2002)

The vertically layered architectures are useful to resolve the above problems partly. The control flows pass through each layer and in the final layer the output is generated in the one-pass control flow architecture. In the two-pass control flow architecture, the control flows go through each layer until the final layer (*one pass*) and they come back to the initial layer (*two pass*) as well as the initial layer generates the output. Both one-pass and two-pass control flow vertical architectures reduce the complexity of interactions between layers. For example, the vertically layered architecture has n layers and all layers have n - 1 interfaces. The maximum no of actions for each layer is m. Finally the interactions between layers is $m^2(n-1)$.

To summarize, let say, the no of layers be 10 and each layer has 10 actions. The maximum no of interactions in horizontally layered architecture is 10^{10} = 10,000,000,000, whereas in vertically layered architecture is $10^2(10-1) = 900$. It is clearly visible that the interactions in vertically layered architecture are much less than that in horizontally layered architecture. For this reason, the performance in vertically layered architecture is increased significantly.

2.2 Multiagent Systems

Multi-agent system (MAS) is a system which consists of a set of heterogeneous and intelligent agents that interact and communicate with each other and interact with their environment for reaching their design goals. The interaction between agents either can be *cooperative* or *competitive*. Every agent receives information from other agents and its environment. The system defines some rules to every agent for making cognitive decisions to achieve its goals. This sections represents the basic on the multiagent systems.

2.2.1 Charactistics of Mulitagent Systems

The characteristics of multiagent systems are discussed in the following [Woo02].

Agent Design: The agents are designed in different multiagent systems in different ways. Different designs of the agents require different hardware and software. If the agent needs different software and hardware, is called *heterogeneous* agent. On the other hand, if the agent is designed with same type of hardware and software, is called *homogeneous*. The heterogeneous agent can perceive its environment better than homogeneous agent. So it can take the possible best decision to get the better outcome.

Control: The control in a single-agent systems is centralized, whereas decentralized in multiagent systems. Decentralized control is always preferable for developing a robust, scalable and fault-tolerant systems. Appropriate *coordination* approaches are required to maintain for interacting the agents with each other.

Knowledge: In the single-agent systems, every agent contains some predefined actions. This agent does not know the effect on its environment after performing its actions. The agents in the single-agent systems cannot understand and realize the adverse behaviors of other agents. On the other hand, the agents in multi-agent systems are designed with artificial intelligence so that they can cope up with diverse behaviors from other agents.

Communication: Multiagent system is a set of interactive agents. The interaction between different agents is connected with *communication*. The communication mechanism is basically two-way process in multiagent systems. All agent can be both *sender* and *receiver* of messages. The communication mechanism can be used for different purposes such as *coordination between cooperative agents* and *negotiation between self-interested agents*. In order to communicate between different agents, network protocols are required to exchange information. The language used needs to be sufficiently expressive to allow agents to transmit complex information and goals, possibly programming each

other [Gen94].

2.2.2 Agent Communication and Coordination

All distributed agents have communication knowledge and functionalities to interact and share information to each other in the mulitagent systems. In dynamic environment, it is difficult to have the complete knowledge of the entire system and all capabilities for an agent. The agents should have standard methods to interact effectively with other agents, understand the environment and use the necessary resources from other agents and environment. Many agent communication languages have been developed to manage communication between different agents. Some of standard languages are described below.

Knowledge Query and Manipulation Language (KQML) is a language and protocol to exchange and share the knowledge between agents in the system. It is the first agent communication language. This language is one of the important parts of the APRA Knowledge Sharing Effort in USA [Sea69]. KQML specification specifies a common format of messages for exchanging between different agents. The message in KQML is treated as an object and each contains a *performative* (class in object-oriented programming) and a set of *parameters*. A simple example of KQML is the following:

```
(
ask-one
:content (PRICE IBM? Price)
:receiver stock-server
:language LPROLOG
:ontology NYSE-TICKS
)
```

Another important and famous language for agent communication is *Agent Communication Language* (ACL) which is proposed by the Foundation for Intelligent Physical Agents (FIPA). It provides the standard methodologies for agent communications. It includes many aspects of KQML [LFP99]. ACL is almost similar to KQML and message format is also similar. The main difference between ACL and KQML is in the collection of *performatives* which they provide [Woo02].

Coordination is an important mechanism in the multiagent systems to execute the activities of the agents. Appropriate coordination between agents can reduce the usage of extra resources, elude the deadlock and sustain the safety critical of the system.

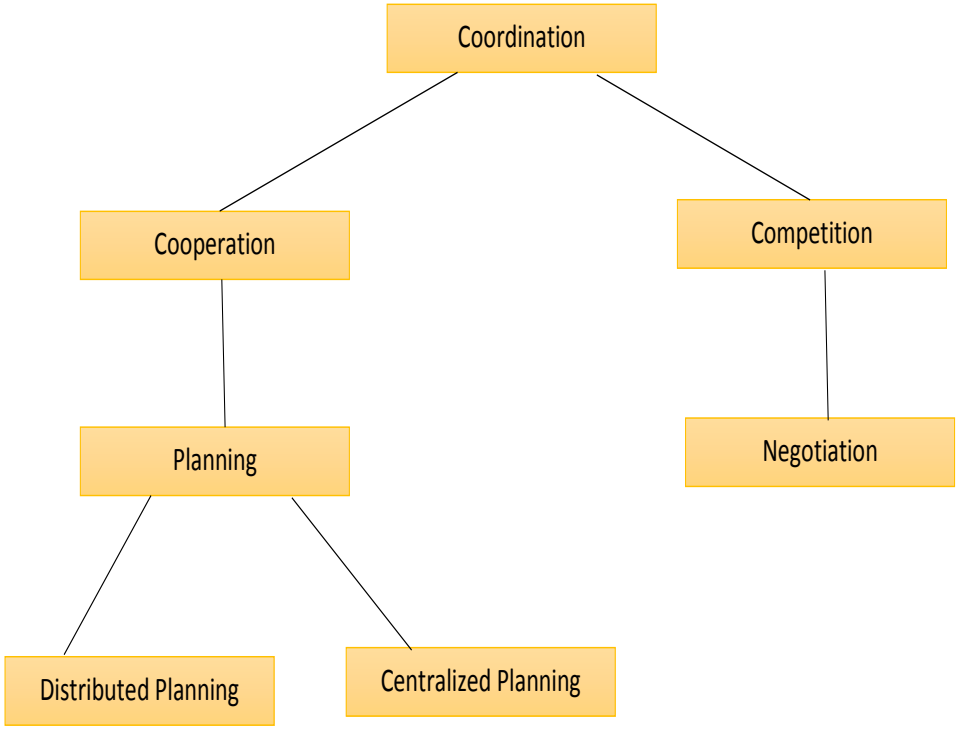

Figure 2.3: Taxonomy of some of the different ways in which agents can coordinate their behaviors and activities.

Source: Michael Wooldridge, (2002)

Figure 2.3 shows that *cooperation* is used to coordinate the *nonantagonistic* agents, whereas *negotiation* is used to coordinate the *competitive* agents. Every agent has to keep the track of other agents to cooperate and communicate them easily. Moreover, it develops a model to maintain the future interactions with each other.

2.3 Agent-based Modeling and Simulation

Agent-based modeling and simulation (ABMS) is a recent technique to analyze and model the dynamic and complex adaptive multi-agent systems with autonomous, self-learning and interacting agents. Nowadays ABMS is very popular for modeling such complex systems. Agent-based modeling is used in many disciplines such as archaeology, biology, ecology to supply chains, consumer market analysis, military planning and economics. Agents have their own behaviors with simple rules. They interact with other agents and influence each other. They learn new behaviors from their environment and experiences. They use their new behaviors and experiences to better adjust to their environment (Figure 2.4). They are self-organized and heterogeneous in their environment. With the

individual modeling of agents, it is possible to realize the whole effects of the diversity among agents in the system as a whole. The diversity exists because of dynamic behaviors and attributes of the agents. Due to many applicable fields of the agent-based modeling, systematic instructions are required on how to develop and apply agent-based models in their appropriate fields. Any typical agent-based model consists of the following three components:

- A set of agents with their attributes and behaviors.
- Relationships and methods of interaction among agents: A network which defines the rules how and whom agents interact.
- The environment of the agents: The agents make their interactions with other agents in their environment.

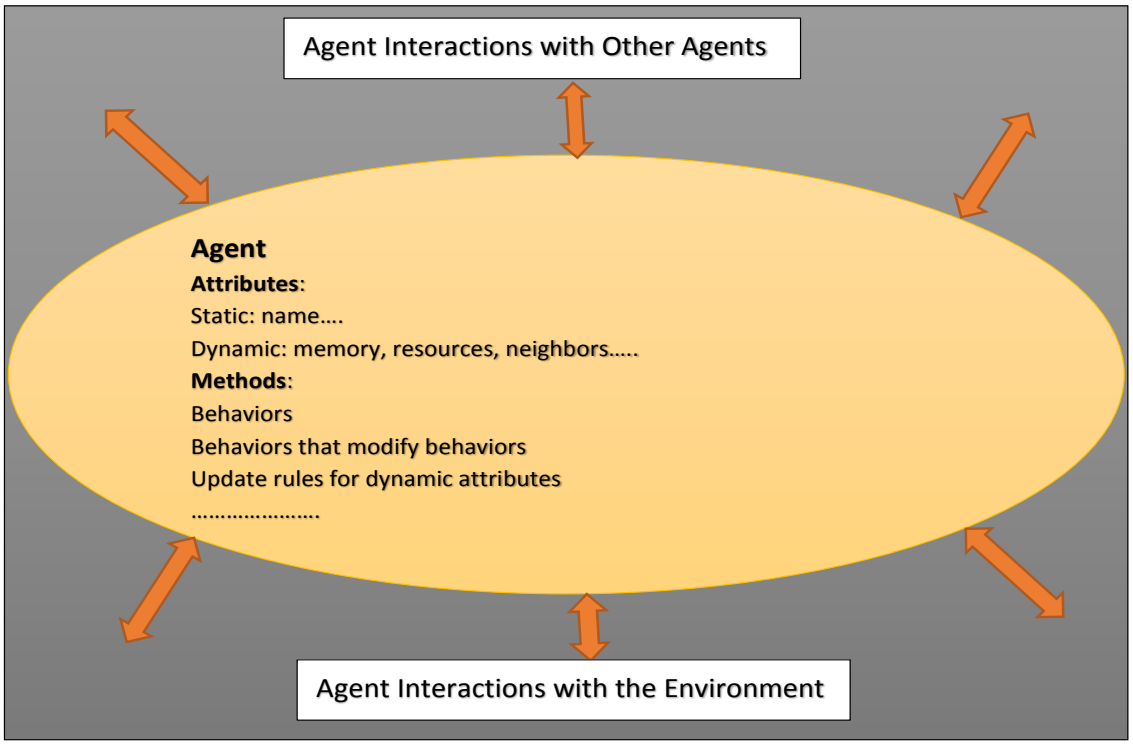

Figure 2.4: A typical agent. Agents have behaviors and interact with other agents and the environment.

Source: CM Macal and MJ North, (2010)

It is essential to analyze, identify and model some important components for developing an agent-based system. For these reasons, every model developer must scrutinize the agents, relationships among agents and environment where agents will interact with each other.

Figure 2.5 illustrates the structure of a typical agent-based model. To perform the agent behaviors and interactions in the simulation; a computational engine, agent-based modeling toolkit, programming language and other required implementations are necessary. The repetition of agent behaviors and interactions are most important to have better result of the agent-based simulation model. Most of the models are developed on the basis of timeline, activity or discrete-event. The agents are able to take the decision independently without any external direction to attain their internal goals.

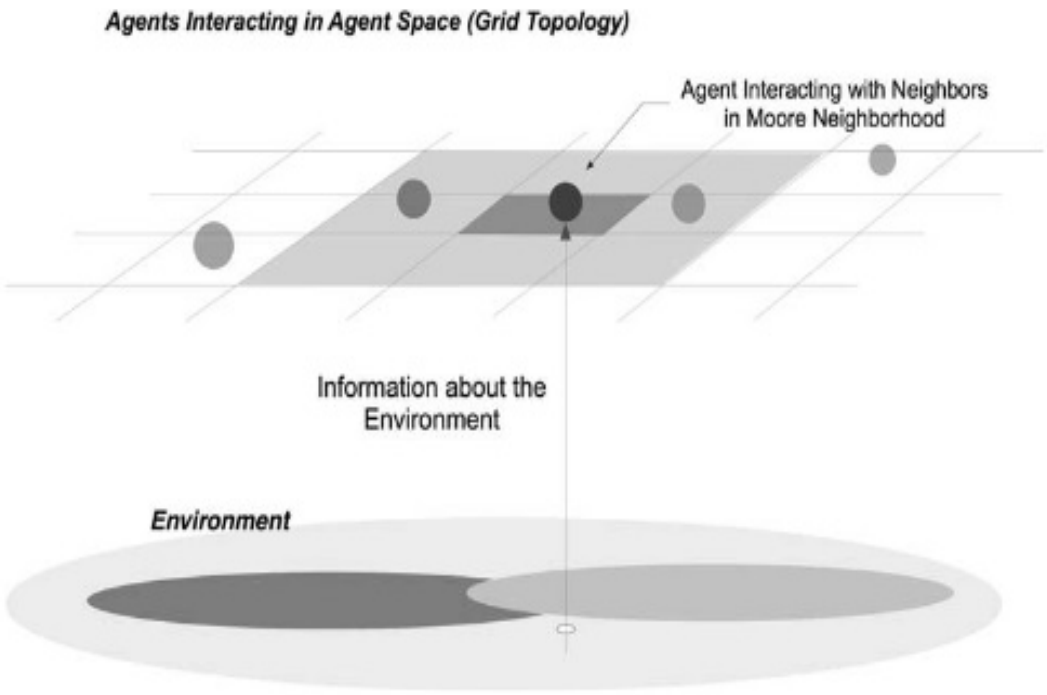

Figure 2.5: The structure of a typical agent-based model, as in Sugarscape.

Source: Epstein and Axtell, (1996)

An agent has the following important features based on how and why agent-based models are developed in various applications:

- An agent is a *self-contained, self-organized* and *unique* identification.
- An agent is *self-directed* and *autonomous*.
- An agent has *state* which varies over time.
- An agent is *social entity* which has dynamic interactions with other agents for influencing its own behaviors.
- An agent is *adaptive*. Adaptability helps the agents to alter their behaviors to adapt in their environment.

– An agent could be *heterogeneous*.

– An agent is *goal-oriented*.

Chapter 3

Overview of AMIRIS Model

3.1 Introduction

Renewable energies are the reliable, affordable, sustainable and environment-friendly energy sources for the future. The main objective of renewable energies is to ensure the greatest contribution to the energy sector in the future. Another goal of them is to substitute the conventional energy sources (i.e. fossil fuels and nuclear energy) to provide steadily a dynamic, affordable and sustainable energy mix. CO_2 emissions and greenhouse effect can be significantly reduced by using renewables instead of conventional energy sources. The uncontrolled expansion of renewable energies can be cause of pressures on nature and landscape. Furthermore, they are intermittent and unpredictable.

Currently, fossil fuels are affordable energy sources in the energy sector. They are very convenient and have significant contribution to energy sector for fulfilling the demand of electricity. Electricity from fossil fuels is less expensive than that from renewables. Because of declining availability of the fossil fuels, they would be expensive in the long run. Additionally, they have high-risks in the environment.

Nuclear power plants have really significant and powerful impact to energy sector. Nuclear energy is a secure and reliable energy source that helps to supply the required amount of electricity. Nuclear power plants can be operated for long period of time. But they are dangerous in case of environmental issues. They can be cause of nuclear accidents that can spread radiated particles around the wide area. Moreover, installation of new nuclear power plants, disposal of nuclear waste and decomposing of nuclear power plants are very expensive.

Nowadays, electricity generation from renewables is a vigorous issue around the world. In order to achieve the stability of electricity generation, top-level research and consis-

tent technological development are needed for sustainable, efficient and low-risky energy systems with renewables. By increasing the proportion of renewables to energy systems, expanding grid and including efficient storage; it is possible to setup one of the firm pillars of the energy sector.

In Germany, renewable energy has prodigious contribution for the electricity generation in the energy sector. The restructure of the electricity system in organizational, technological and financial aspects is essential to produce the electricity with the expansion of *renewable energy sources* (RES) according to the political targets of the German government [BMU10]. *Renewable Energy Source Act* (EEG) has defined policy framework for the implementation of renewables into the energy system. The main focus of EEG act is to move from conventional energy to renewable energy for securing the future energy sector. EEG has decided to produce about 80% electricity from renewable energy by 2050 to maintain the demand of electricity.

Agent-based modeling (ABM), instigating from the field of artificial intelligence, is an approach to analyze and investigate such complex and multi-linked systems with independent actors [Tro09]. AMIRIS (Agent-based Model for the Integration of Renewables Into the Power System) is an agent-based model which is developed to analyze direct marketing and market integration of renewables.

Section 3.2 describes policy framework of green electricity privilege, feed-in tariff (FIT) as well as direct marketing and market integration of renewables. Section 3.3 analyzes different agents (i.e. actors) of AMIRIS model, and the behavior and roles of agents and describes different analyzing methodologies. Section 3.4 presents an overview of the architecture and functionalities of the AMIRIS model.

3.2 Policy Framework

German government has liberalized the German energy markets in 1998. Because of liberalization, many agents (actors) are involved to the energy markets. In order to balance the demand and supply of the electricity, it is very crucial to have a reliable system. Some renewable resources such as solar and wind power are intermittent, volatile and unpredictable. The energy systems and markets will be responsible to maintain intermittent and unpredictable renewable resources in the future because of insecurity of their availability. Biomass, geothermal, water, energy storage, grid expansion and the management of demand can help to equilibrate between the demand and supply of the electricity.

Renewable Energy Source Act (EEG) was the most important appliance for assisting the RES in the previous years. The electricity, produced by power plant operators, was sold with *buy-offs* guaranty and *feed-in tariffs* (FIT) through *Transmission System Operators* (TSO). Due to the EEG, new suppliers, intermediaries and new technologies are included to the energy markets. The EEG was released in 2000 and revised many times in between 2004 and 2009. Finally, it was reformed with some important modifications in 2012 to support the direct marketing of renewable energy.

'Optional market premium' was an important inclusion in the revision of EEG act in 2012. It enables to transfer the duties of selling renewable energy from the TSO to renewable power plant operator (PPO) or intermediaries respectively and leads to the development of new business models, innovative direct marketing strategies and new forms of cooperation between actors.

3.3 Actor Analysis

It is prerequisite to analyze the relevant actors and their roles for developing the agent-based simulation model. For the AMIRIS model, the sociological theory of strategic action fields and concepts of neo-institutionalism of organizational sociology were used to analyze the actors. The theory of strategic action fields [Fli11] offers a specific viewpoint by interpreting activities related to direct marketing as the attempt of competing actors to shape and design a specific field of action as a new market. There are typically three classes of actors in emergent field. They are incumbent, challengers and governance body. The roles and strategies of the actors depend on the actual state of a field, which can be a) rather unorganized and emergent, or b) organized, stable and only slightly changing, or c) organized, unstable and open to transformation [Fli11].

Actors should be competitive and have their identities. It is very difficult to distinguish an actor without its identity. In our model, neo-institutionalism organizational theories are used to analyze the challenging interests and identities of the corresponding actors. Sociological concepts have established alternative approaches to the typical neo-classical understanding of actors like homo economics. Different actors have different behaviors, backgrounds and environments. They follow some important strategies to achieve their desired goals. Actors, interactions among actors, coalitions, business models and policies are defined as phenomena of an emergent strategic action field of the electricity system with the relationship of the direct marketing of electricity from renewables. Moreover, numerous actors define the rules to compete each other. Although the roles of all actors (incumbent, challengers, governance units) are clearly defined, since the actors do not

have sufficient knowledge of each other they cannot predict the future development properly. Neo-institutionalism in organizational sociology is used to analyze different actors' strategies, interests and identities to solve the above problems. The incumbent actors have developed coal and large fossil fuel power plants and central power plants such as nuclear power plants. Due to huge subsidies for coal and nuclear energy technologies on the one hand, and the externalization of environmental and social costs on the other, incumbent actors and fossil energy technologies have profited from non-transparent prices and asymmetric competitive advantages [Mil02]. The incumbent energy technologies have received direct and indirect subsidies for decades [Jac04].

The challenging actors (new actors) have developed new technologies to produce the electricity from renewables. Technological developments in photovoltaic, wind turbines, biogas technologies etc. had been advanced by environmentally-driven scientists in young research institutes, by new firms and environmental groups, and by lead users, also driven by green ideas and therefore supportive of renewable energy technologies [Jac06] [Ohl08] [Fuc12].

Political actors have very vital roles in the German electricity system. They should provide a suitable environment for marketing of these new technologies. Throughout the 1990s associations, local groups and societies were founded with the aim of improving and enhancing political support for the infant technologies and their commercialization: it was a coalition of various, mainly new actors that managed to influence the federal government to develop innovative policy instruments designed to support the expansion of renewable energy technologies [Fuc08] [Ohl08].

EEG is the most important appliance to establish the renewablesto in the electricity energy system. Still now it is required discussion and amendment on EEG to provide a niche market for the new technologies and setup a stabilized energy system. It defines the priority to electricity from renewable energies to store into the grid operators and guaranties fixed feed-in tariffs.

In order to implement the new successful products, business models and routines in the strategic emergent field; coalition formation and institutionalization have crucial importance. Any kind of analysis of the emergent field must identify and describe the relevant actors according to their origin, history, traditional links, size, power and control. Organizations are not only efficiency-oriented, but also cope with environmental expectations, developing various legitimacy strategies in the process [Has05].

3.4 The Structure of the AMIRIS Model

AMIRIS (Agent-based Model for the Integration of Renewables Into the Power System) is an *agent-based modeling* (ABM) simulation system which is developed to:

- Scrutinize the action patterns and mutual interactions at the micro (actor) and macro (system) level to better understand the organization of markets for the further expansion of Renewable Energy Sources (RES).
- Analyze the impact of the design of support mechanisms on technical and market integration of RES.
- Analyze the direct marketing of renewables.
- Investigate the potential sources of revenue for RES in systems with high RES shares and the conditions for their use.
- Analyze the investment conditions for renewable energy plants when being fully integrated in the electricity markets.

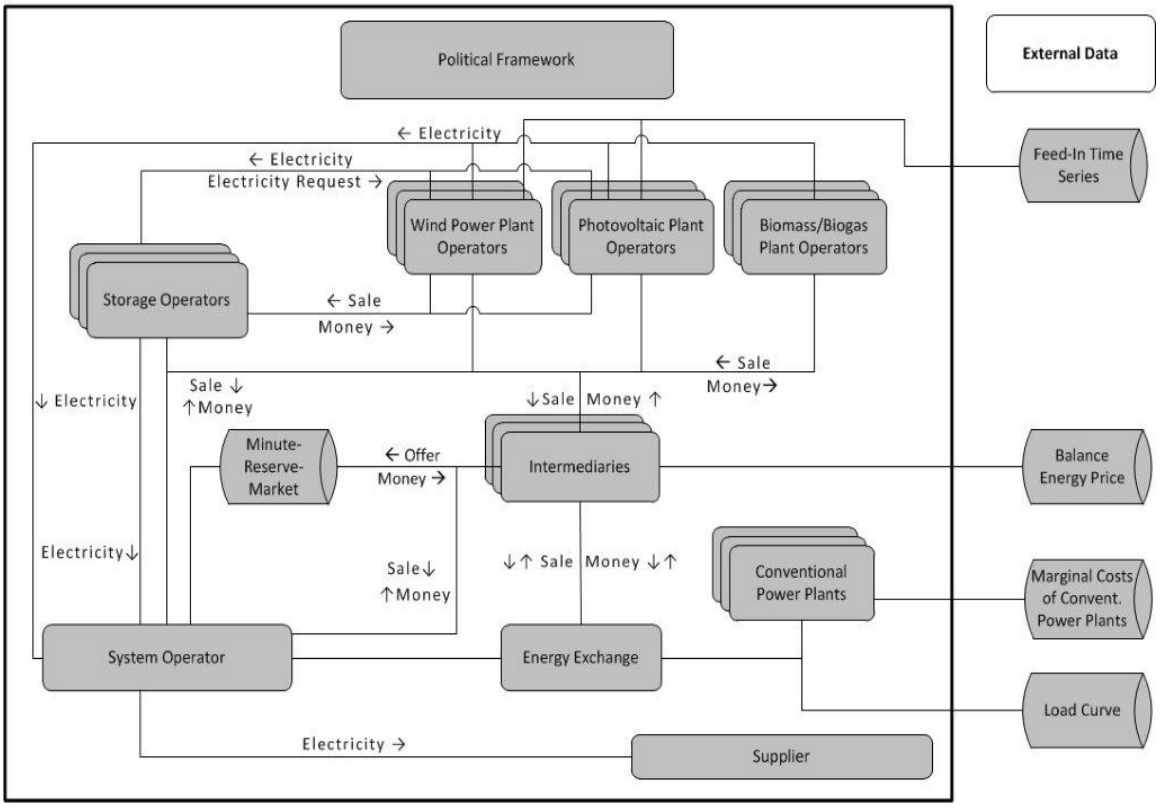

Figure 3.1: Structure of the AMIRIS Model. Description of activities and interactions among different actors.

Source: AMIRIS, Institute of Technical Thermodynamic, DLR.

Figure 3.1 represents the overall structure of the current AMIRIS model. The renewable energy plant operators can either provide the produced electricity directly to the grid operator ("passive" plant operator) with *feed-in-tariff* (FIT) or make contracts with different intermediaries and sell their electricity ("active" plant operator) to the intermediaries.

The intermediaries take all responsibilities and obligations of marketing, and get in return from the grid operator compensation payments of funding instruments (market premium). The intermediaries pay to the plant operators with a bonus or a portion of their marketing profits. As an additional source of income the intermediaries can offer certain classes of biomass plant operators in the balancing energy market.

Physically, the electricity always flows through the Grid Operator to the supplier, which symbolizes the demand and serves as a sink of the system.

The AMIRIS model can simulate over several years, whereas for every hour the electricity production, its selling and the predicted production are calculated. A single run in AMIRIS model is a set of multiple individual executions. An execution is a complete calculation process for certain years which defined by the user.

Chapter 4

Parameterization and Batch Processing

4.1 Introduction

Repast Simphony is an agent-based simulation toolkit for developing agent-based models across a variety of domains and applications. It has been used to develop the AMIRIS model. Repast Simphony has some important features such as Parameterization, Batch Processing, Scheduling, and Parallel Processing etc. Parameterization and Batch Processing are the main concern for this thesis. Parameterization is one of crucial parts of Repast Simphony Toolkit. Parameters are used to define different criteria of a model such starting point, ending point, interval of method execution of the model. There are two categories of parameterization in AMIRIS model.

Further investigation and analysis are required to develop a dynamic and flexible parameterization mechanism for defining the model criteria and for initializing the agents with relevant parameters in dynamic AMIRIS model.

Section 4.2 describes usages and assessment of different data storage applications (i.e. Excel, Access and XML) for managing the parameters to define the model criteria and initialize the agents. The selection criteria of appropriate storage application for the AMIRIS model are also discussed in this section. Section 4.3 discusses the validation mechanisms for the parameters. Section 4.4 introduces different parser mechanisms to parse the parameters from the selected data storage application.

Most of the simulation models need the iterative execution with different parameters for statistical analysis. In order to execute an agent-based simulation model repeatedly, job processing or batch processing should be implemented for the AMIRIS model. Repast

Simphony has some useful and sophisticated features to develop batch processing for executing a model multiple times in a single run. Section 4.5 illustrates batch processing mechanism which is used for iterative execution of the model.

4.2 Storage Mechanisms of Input Parameters for AMIRIS

Nowadays data storing and management are the important issues in the software world. Several storage applications exist to store and organize data such as Microsoft Excel, Comma Separated Values (CSV) file, Microsoft Access, Relational Databases, Extensive Markup Language (XML) etc. Different criteria are considered for different software programs to select an appropriate storage application according to the requirements of the software. In Table 4.1 some of the important criteria of storage application are listed which are deliberated to opt an appropriate data storage application for the input parameters of the AMIRIS model.

List of Considered Criteria
Platform Independency
Application Independency
Open-source Facility
Processed Data by Software Program
Validation
Input Restriction
Java Parser Facility
Connection Orientation
Concurrency Control Mechanism
Flexibility
Human Understandability
Computer Understandability
Scalability
Identical Characteristic
Standardization of Input Data
Data Analysis

Table 4.1: List of criteria for the selection of data storage application

Platform Independency: It means that the data storage application is not bound to run on a particular operating system. The storage application should be compatible with all operating environments. As the AMIRIS model is developed with Java which is a cross-platform compatible programming language, the data storage application should be platform-independent to execute it on different operating systems in case of future necessity.

Application Independency: The storage application should be application independent so that it can be reused if the AMIRIS model is required to switch to another programming language.

Open-source: It means the storage application is free of cost. No license is required to buy to use it. Due to cost optimization, the storage application should be open-source for the parameters of the AMIRIS model.

Processed Data: Most of the storage applications provide normal text data to the software programs after processing the data so that the data is readable by the software. Processed data with desired data types is useful and convenient for the software programs. In order to get the typed data, the AMIRIS model requires a storage application which supports primitive, user-defined and other useful data types.

Validation: Validation defines how data is structured in the storage application and assures clean, correct and useful data to the software programs.

Input Restriction: It means that the particular input data must be in a certain range or format. For example, one value of a share of a business in percentage must not be less than 0 and greater than 100. A storage application with this feature is important for AMIRIS model to optimize the code structure and to omit faulty parameterized runs.

Java Parser: It is a power tool to parse data with a structured way from storage applications. The AMIRIS model requires such a storage application which supports Java parsing mechanism, since it has been developed with Java programming language.

Connection Orientation: Most of the data storage applications require a connection with the software programs to fetch the data which is time consuming. A connection-less data storage saves the time which is an important issue for the AMIRIS model.

Concurrency Control Mechanism: Concurrency is a mechanism to access the data by the users from a storage application simultaneously. Concurrency control mechanism is required in case of parallel processing of AMIRIS model.

Flexibility: Flexibility of creating data file, adding new data and modifying existing

data is an important concern for choosing a storage application.

Human Understandability: The data structure of the storage application should be easy and simple so that the user can easily understand the exact meaning of data which is considerable issue to select a storage application for the software programs.

Computer Understandability: A storage application should be easy to comprehend by the software programs for interchanging the data in a well-formatted manner between them. AMIRIS model does not only fetch data from the storage application but also stores the necessary data to that storage application. For this reason, the AMIRIS model requires a comprehensive and flexible storage application.

Scalability: Every software program requires a suitable and efficient storage application to optimize the performance. Although currently the amount of the input parameters is small, it can be increased in the future. For this reason, the AMIRIS model needs such a storage application which is suitable for both small and large amount of data.

Identical Characteristic: Most of the storage applications store data in an identical manner. Since the input parameters of AMIRIS model are the combination of both identical and non-identical data, the AMIRIS model requires a storage application which can store both types of data with a well-structured format.

Standardization of Input Data: The standardization of input data regarding storage applications means that every written language by human should be supported for the input data. The AMIRIS model needs a storage application which supports international standard for the input parameters.

Data Analysis: Data analysis is a process for scrutinizing, transforming and modeling the data to achieve useful and meaningful information. Data analysis is an essential part for most of the simulation models. Since the AMIRIS is a simulation model, it might require a storage application which provides the data analyzing features and/or supports data analyzing tools.

This section describes and evaluates some of popular storage applications in accordance with the criteria which are defined above.

4.2.1 Microsoft Excel

Microsoft Excel is an electronic spreadsheet application which is developed by Microsoft Corporation. Normally it is used for storing, organizing, manipulating and analyzing large amount of data. In addition, it is a powerful tool for graphical and statistical

analysis of data, making pivot tables as well as macro programming language named as Visual Basic for Applications. The data types in Excel are normally numbers, text or formulas. Microsoft Excel stores the data in a single table or worksheet called *flat* or *non-relational* data.

Some suitable domains and scenarios of usage of Microsoft Excel are illustrated in the following:

Accounting: Microsoft Excel has the powerful and sophisticated techniques and features for financial data analysis. The important usages of Excel are to create cash flow statement, transaction statement, profit and loss statement as well as income statement in the accounting section of an organization.

Budgeting: Defining and creating budget is vital issue for an organization. Excel provides some powerful features to create an optimal budget. For example, a marketing budget plan, an event budget, or a retirement budget.

Billing and Sales: Every company and organization maintain their billing and sales data. They need to create the required forms for billing and sales data. It is easy and simple to create these forms using Excel. For example, sales invoices, shipping slips, packing slips, or purchase orders.

Reporting: Excel is very sophisticated tool to generate reports. Most of the organizations use it to analyze and summarize their important data. For example, reports that measure project performance, forecast data, summarize data, or present variance data.

Planning: The activities of an organization depends on its plan or on its planners. Excel is one of the professional planning tools. It is used to make a weekly class plan, a marketing research plan, a year-end tax plan, or planners that help you plan weekly meals, parties, or vacations.

Tracking: It is also useful to keep track of data in a time sheet or list. A time sheet for tracking work and an inventory list that keeps track of equipment are the appropriate examples.

Using calendars: Because of its grid-like nature, Excel lends itself well to creating any type of calendar. For example, an academic calendar to keep track of activities during the school year, or a fiscal year calendar to track business events and milestones.

Table 4.2 shows a simple example how tabular data are represented in Microsoft Excel. This data table contains some information of different agents. It consists of three columns named as `ID, Agent Name` and `Agent Type`. The first row is called header which adds

column names. The following rows contain the same number of values like header row. The `ID` and `Agent Name` columns contain the values to identify every agent and the values of the agents' name respectively. The `Agent Type` column stores the values to recognize the type of agent whether it is *producer* or *buyer*. This table is an example of identical data representation.

ID	Agent Name	Agent Type
WAB_11	WAB	Producer
WAB_21	WAB	Producer
PvAB_11	PvAB	Producer
PvAB_21	PvAB	Producer
BmAB_11	BmAB	Producer
BmAB_21	BmAB	Producer
ZWH_1	Zwischenhaendler	Buyer
ZWH_2	Zwischenhaendler	Buyer

Table 4.2: Representation of Tabular Data in Microsoft Excel

Table 4.3 illustrates the summary of the defined criteria in Table 4.1 for Microsoft Excel as a suitable storage application.

Microsoft Excel is compatible only for Microsoft Windows and Mac operating systems. Moreover, it is not compatible with all software applications. Since Microsoft Excel is not an open-source application, the users need to spend money for getting the license to use it. Software programs get the processed data as normal text. Data validation is an important concern for most of the software programs. But Excel does not provide any validation feature or even support any validation mechanism for the data. Furthermore, it cannot restrict the input data in a certain range or format as well. Java programming language has been used to develop the AMIRIS model, but no parsing mechanism in Java is available for Microsoft Excel. Software programs do not require any connection to Microsoft Excel for fetching the data. They retrieve data by using the location path of the Excel file.

Although Excel is a popular and useful storage application, it does not serve as a database management system and does not provide any concurrency control mechanism. Excel file is considerably easy to create and understand by users, but difficult to understand by the software programs. Microsoft Excel is suitable for both small and large amount of identical data and supports most of the languages which are written by human. Data analysis is an important issue for most of the simulation models. Microsoft Excel is a

powerful and efficient tool for statistical and graphical analysis which is mentioned earlier.

Storage Application / Criteria	Microsoft Excel
Platform Independency	Dependent
Application Independency	Dependent
Open-source Facility	Not open source
Processed Data by Software Program	String
Validation	Not possible
Input Restriction	Not possible
Java Parser Facility	No
Connection Orientation	No
Concurrency Control Mechanism	Does not support
Flexibility	Flexible
Human Understandability	Easy
Computer Understandability	Difficult
Scalability	Not scalable
Identical Characteristic	Good for identical data
Standardization of Input Data	International
Data Analysis	Good

Table 4.3: Summary of criteria for Microsoft Excel

4.2.2 Comma Separated Value (CSV) Files

The short for Comma Separated Values, CSV file format is frequently used to interchange and convert tabular data among similarly different applications within short time. It is possible to differentiate the fields using any special character. For this reason, it is also called character separated values. CSV file stores normally numeric values and text in plain-text format. It was very much popular before invention of Extensible Markup Language (XML). Due to XML, CSV file has become outdated for data exchange possibilities. Normal text editor, Microsoft Excel, OpenOffice spreadsheet applications can be used to create CSV files. CSV file contains two type of rows. The first row is called header row which contains the column names. The header cannot contain escaped characters and

apostrophes. The following rows contain the same number of values like header row. Each row is a record which must be ended in a line. Every individual field is separated by a single comma. A CSV file can store any number of records. The CSV file format does not have any standard. RFC 4180 has just described some fundamental rules. Although it is frequently used by scientists, researchers in various applications and domains, the rules of CSV file are not well-documented. CSV files are practically quite portable, although it is not a well-organized format.

Many applications, for instance, research, scientific, business and web applications, use CSV file because of common and simple file format. Data exchange between different applications is one of the common usages of CSV file. Sometimes many applications require to transfer data from one format to another. For example, an application have to transfer data from its relational database to a spreadsheet or text file. Most of the database applications have such features to export data as CSV file format. After exporting data by these applications, the spreadsheet program or text file can then easily import the CSV file.

RFC 4180 defines the format for the "text/csv" MIME type registered with the IANA. (Shafranovich 2005) Another relevant specification is provided by Fielded Text. Creativyst (2010) provides an overview of the variations used in the most widely used applications and explains how CSV can best be used and supported.

The typical rules of specifications and implementations of CSV file format are described below:

- As CSV file is a delimited data format, each column must be separated by comma or special character or string, and each row ended with newline.
- A CSV file does not require a specific character encoding, byte order, or line terminator format.
- A line terminator can be part of a column/field. Since it is used to separate rows/records, sometimes it is very difficult for some software programs to identify if the line terminator is used as input into the field.
- All records should have the same number of fields, in the same order.
- All data are represented as a character sequence instead of a byte sequence. For instance, a numeric value 1000 and a text value 'AMIRIS' both are interpreted as a sequence of characters.
- Head-to-head fields must be separated by a single comma. They can be separated by semicolon, tab or other special characters. For example: 1, WindPowerPlant, Producer

– The fields that have comma, line breaks etc. inside the input must be quoted. For example: 1, "Wind, Solar Power Plant", Producer

Table 4.4 shows a simple example of data representation in CSV file.

ID, AgentName, AgentType
1, WAB, Producer
2, WAB, Producer
1, PvAB, Producer
2, PvAB, Produer
1, BmAB, Producer
1, Zwischenhaendler, Buyer
2, Zwischenhaender, Buyer

Table 4.4: Representation of Tabular Data in CSV File

This table includes some information of the agents. It consists of three fields. The fields are *ID, AgentName* and *AgentType*. The first row is known as header row like Microsoft Excel. The header row consists of column names of the CSV file. Like Microsoft Excel, the following rows contain the same number of values according to the header row. The *ID* and *AgentName* columns include the values to identify every agent and the values of the agents' name respectively. The *AgentType* column holds the values to recognize the type of agent whether it is *producer* or *buyer*.

Table 4.5 summaries the defined criteria in Table 4.1 for CSV as an appropriate storage application.

Comma Separated Values (CSV) file is a platform and application independent storage application. All applications and operating systems can easily interchange data between them by CSV files. Moreover, CSV is an open-source storage application and the user does not need to spend money to get the license. Like Microsoft Excel, the processed data by the software program is normal text. Software program itself needs to process data for getting the expected data. In order to generate the expected output, validation is very important issue of the software programs. Unfortunately, CSV does not have any validation feature or even does not support any external validation mechanism for the data. Moreover, since CSV stores data as text, it cannot restrict the input data in a certain range. The AMIRIS model has been developed by Repast Simphony which is Java-based simulation toolkit. But Java programming does not provide the parsing facility to process data from CSV files.

Storage Application / Criteria	Comma Separated Values (CSV)
Platform Independency	Independent
Application Independency	Independent
Open-source Facility	Open source
Processed Data by Software Program	String
Validation	Not possible
Input Restriction	Not possible
Java Parser Facility	No
Connection Orientation	No
Concurrency Control Mechanism	Does not support
Flexibility	Flexible
Human Understandability	Difficult
Computer Understandability	Difficult
Scalability	Not scalable
Identical Characteristic	Good for identical data
Standardization of Input Data	International
Data Analysis	Not good

Table 4.5: Summary of criteria for CSV

To retrieve data from CSV files, software programs do not require to make a connection to the CSV files. The software programs just use the location of the CSV file. Even though CSV is a popular and portable storage application to interchange information between heterogeneous applications and operating systems, it does not serve as a database management system and does not have the concurrency control facility. Comparatively it is easy to create CSV files. But it is difficult to process CSV files by the computer programs. CSV file can store any amount of identical data and supports most of the written languages by human. In order to realize the impact of the output, statistical analysis and graphical representation are the important issues of most of the simulation models. Although CSV does not provide any analysis tool, it is compatible with Microsoft Excel, and can use analysis tools and features of Microsoft Excel to analyze the data.

4.2.3 Microsoft Access

Microsoft Access, a relational database management system (RDBMS), is developed by Microsoft Corporation. It is a member of Microsoft Office which has many similarities with Microsoft Excel. It can also store large amounts of data like Microsoft Excel. Furthermore, it is able to run powerful queries and analysis tools to slice and dice that data, and perform sophisticated calculations that return the expected data to the user. Another important thing is that if the users store data into Microsoft Access, they can analyze that data using Microsoft Excel as well. Access stores the data into single or more tables which are called relational data. Generally it is useful for better management of data, availability for multiusers concurrently, easy searching and keeping data organized.

Now the question is when Microsoft Access is better to choose as a data storage? Access would be better choice, if the users want to maintain and record the data as well as have to display, export and/or print some part of or whole data simultaneously. The users can define data types in Access what data they require like integer, float, double, string and character etc. In addition, they can identify every record uniquely using primary key. It is possible to make relationships between different tables by foreign keys.

It is really tough to define the actual domains and application area of Microsoft Access. Different type of applications can be built with Microsoft Access. Some suitable domains and applications of usage of Microsoft Access are:

Contact management: The users can maintain contacts and mailing addresses and personal information of individuals to create reports. They can make form letters, envelopes and mailing labels by merging with Microsoft Office.

Inventory and Asset Management: The users can store the information of products of their business. Microsoft Access is very useful to keep information of different product groups.

Order Management: They can manage information of products, customers, orders, shipments, sales, and revenue. Then they can analyze sales, revenue by employee, country, time duration or other important criteria.

Task Management: The users can track the tasks of their employees. They can assign new tasks to the employees and update existing tasks. They can assess the performance of their employees by creating monthly or annual report.

Library Management: Microsoft Access is a powerful tool to organize libraries. It can keep the track about books, magazines, articles etc. The librarian can use Microsoft

Access to track who have lent book, magazines etc.

Event Management: It is also useful to manage different events, keep information of event dates, locations, and number of participants and their personal data. Then the users can easily schedule and organize the events.

Table 4.6 demonstrates the summary of the defined criteria in Table 4.1 for Microsoft Access as an appropriate storage application.

Storage Application / Criteria	Microsoft Access
Platform Independency	Dependent
Application Independency	Dependent
Open-source Facility	Not open source
Processed Data by Software Program	String
Validation	Not possible
Input Restriction	Not possible
Java Parser Facility	No
Connection Orientation	Yes
Concurrency Control Mechanism	Support
Flexibility	Inflexible
Human Understandability	Difficult
Computer Understandability	Easy
Scalability	Scalable
Identical Characteristic	Good for identical and non-identical data
Standardization of Input Data	International
Data Analysis	Good

Table 4.6: Summary of criteria for Microsoft Access

Like Microsoft Excel, Microsoft Access can perform only on Microsoft Windows and Mac operating systems and it is not also compatible with all applications. So it is a platform and application dependent storage application. Furthermore, the user needs to buy license to use it. This means it is not an open-source storage application. Microsoft Access is a powerful and efficient storage application to store and manage data. But it provides only normal text data to software programs and the software programs need to process data to get the expected ones. Although data are stored with primitive data types in

Microsoft Access, Microsoft Access does not provide validation of the data when software programs process the data from Microsoft Access. As mentioned earlier, Microsoft Access can specify data with required data types but it cannot restrict the input data in a certain range. Java programming language has been used to build the AMIRIS model but it does not provide the parsing mechanism for retrieving data from Microsoft Access. Moreover, establishment of a connection between a software program and a data storage application needs extra time which can degrade the performance of a software program. Software programs require a connection to Microsoft Access to fetch data. The user should have sufficient knowledge to store and manage data in Microsoft Access. Different software programs can simultaneously access data from the same database. Sometimes it is difficult to create a database and manage data of the database into Microsoft Access for a new user. On the other hand, it is easy to access data from Microsoft Access by software programs though the processed data is normal text. Microsoft Access is a useful tool for storing both identical and non-identical data and can also store and manage large amount of data. It supports most of the written languages by human. By incorporating Microsoft Excel, Microsoft Access can manage statistical analysis and graphical representation of the data.

4.2.4 Relational Database (MySQL, Oracle, Microsoft SQL Server)

The short form of Relational Database Management System, RDBMS is a database management system (DBMS) based on relational model, and basis for SQL (Structured Query Language) and all modern database systems. It is introduced by E.F. Codd. There are many RDBMSs to store data as a collection of tables with two-dimensional array. Some of popular and modern Database Systems are MySQL, Oracle, Microsoft SQL Server, and IBM DB2. This section describes only common features of relational database systems.

Relational model is a conceptual database model of database management system for structuring data using relations. Every table contains rows and columns as well as header and body. The header consists of column names and the body comprises of the data set. A simple example to demonstrate some tables of a relational model is the following.

```
agent (id, name, type)
communication (id, agentId, agentClass, active)
akteurSE (id, agentId, type, riskType)
```

In this example every table contains a *primary key* which is used to identify the input row. Moreover, *communication* and *akteurSE* tables include *foreign keys* to establish the

relations with other tables. The underlined and bold attributes represent primary keys, and the underlined and non-bold attributes represent foreign keys in the tables.

Figure 4.1 shows a graphical representation of a relational model.

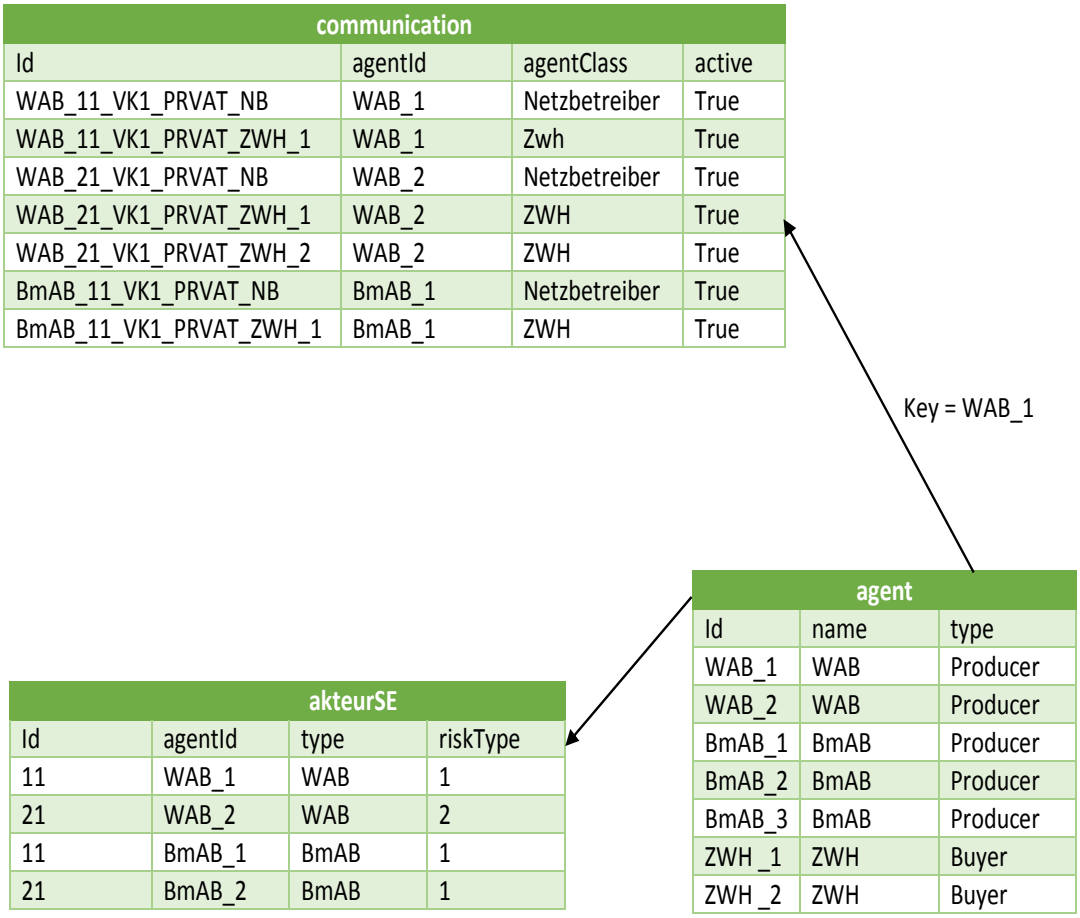

Figure 4.1: An example of a relational model.

Source: Author's illustration

Different tables store different types of input data according to their design structures. The table *agent* contains information of different agents with distinct identity, name and type of agent. *communication* table holds contact information between different agents. The attribute *agentId* in *communication* table is a foreign key which is used to establish a relation between *agent* and *communication* tables. By using this foreign key, the user can easily retrieve contact information of an agent from *communication* table. For example, the key value ("WAB_1") can be used to fetch the contact information of the agent that has the value of id is `WAB_1` from *communication* table. On the other hand, *akteurSE* table comprises information of different agents with risk-type of the agents. Likewise,

a*gentId* attribute is a foreign key which makes a relation between *agent* and *akteurSE* tables to get information of the corresponding agent.

Structured Query Language (SQL) is a database language which is used to store, manipulate and fetch data into and from relational database. SQL is used as a standard database language by all modern relational database management systems. Some important reasons to use SQL are the following.

- Data access from database.
- Data insertion into the database.
- Describing the data.
- Data update in the database.
- Data deletion from the database.
- Embedding within other languages using SQL modules, libraries and pre-compilers.
- Creating and dropping databases and tables.
- Creating view, stored procedure, functions in a database.
- Setup permissions on tables, procedures, and views.

SQL Process is a mechanism to retrieve data from database system using SQL commands. When RDBMS receives an SQL command from a user, it identifies command type and determines the optimal way to interpret the command. Finally, it returns the result of given command to the user. SQL process consists of Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Figure 4.2 represents an example of a SQL process architecture.

SQL Query: The user writes and executes the SQL command query into SQL query analyzer. For example, `SELECT * FROM agent`. SQL query is case-insensitive.

Query Language Processor: It stores the input query to process and generate the expected results.

Parser and Optimizer: Parser reads the command statement and breaks into keywords (i.e. SELECT, FROM), expressions, and identifiers. On the other hand, optimizer analyzes different mechanisms to provide the best expected results to the user.

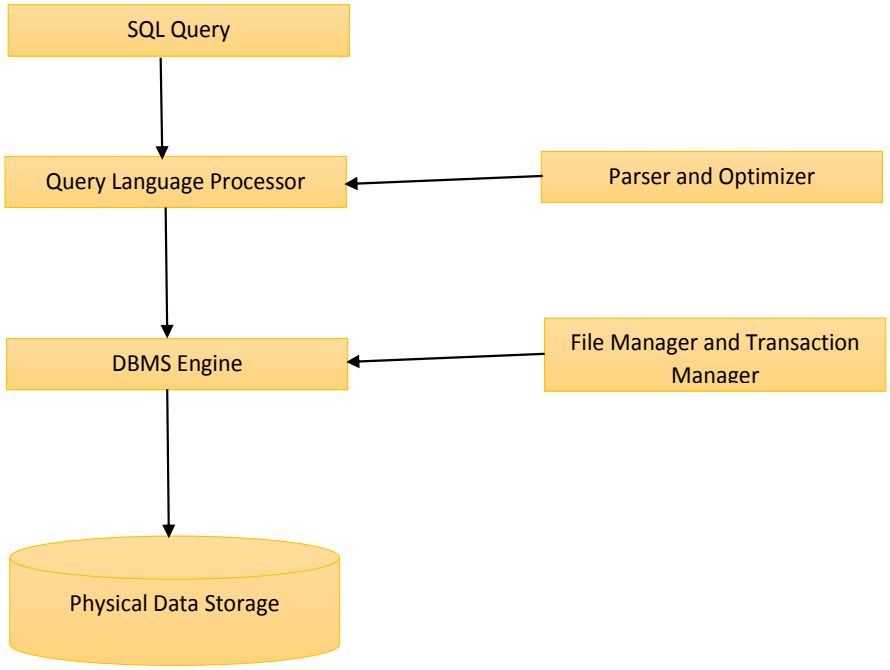

Figure 4.2: An example of SQL Process Architecture.

Source: http://www.tutorialspoint.com/sql/sql-overview.htm

DBMS Engine and Physical Device: It processes input command and sends the request to physical storage. The physical device returns the result to relational engine according to the request. Finally, the relational engine returns the result to the user. The fetched results are as follow from *agent* table (see Figure 4.1).

Id	Name	Type
WAB_1	WAB	Producer
WAB_2	WAB	Producer
BmAB_1	BmAB	Producer
BmAB_2	BmAB	Producer
BmAB_3	BmAB	Producer
ZWH_1	ZWH	Buyer
ZWH_2	ZWH	Buyer

File Manager and Transaction Manager: When the SQL query is executed successfully, transaction manager commits the execution and file manager generates a log file to keep the track of the transaction.

Database systems are widely used in different areas because of their numerous advantages. Some common database applications are listed here.

Airlines and railways Management: Databases are very popular and powerful tools to manage the reservation and schedule information of airlines and railways.

Bank Management: Banks use databases to maintain and process the accounts, loans, inquiries and transactions. The bank management can analyze the revenue and evaluate the performance of employees.

Education Management: Schools and colleges use databases for course registration, result, and other information.

Telecommunications: Databases are used in telecommunication sectors to store and manage the communication network information, telephone numbers, record of calls, monthly bills of customers etc.

Credit card transactions: Banks use databases to keep track of purchases on credit cards. They process the tracking information to create the monthly transaction statement.

E-commerce: E-commerce use databases to store and integrate the heterogeneous information sources. For example, online shopping, consulting a doctor or holiday package booking.

Health care information systems and electronic patient record: Hospitals and clinics use databases are for storing the personal information of patients and maintaining the patient health care details.

Digital Library Management: Databases are used for management and delivery of large amount of textual and multimedia data.

Finance: Finance organizations use databases for storing and analyzing sales, purchases of stocks, annual revenue information.

Sales: Databases are used to store product, customer and transaction details.

Human Resource Management: Databases are used in many organizations to store their employees, salaries, benefits, taxes information, and they are also useful for generating salary checks.

Table 4.7 demonstrates the summary of the defined criteria in Table 4.1 for relational database as a suitable storage application.

Storage Application / Criteria	Relational Database
Platform Independency	Not all databases
Application Independency	Not all databases
Open-source Facility	Not all databases
Processed Data by Software Program	String
Validation	Not possible
Input Restriction	Not possible
Java Parser Facility	No
Connection Orientation	Yes
Concurrency Control Mechanism	Support
Flexibility	Inflexible
Human Understandability	Difficult
Computer Understandability	Easy
Scalability	Scalable
Identical Characteristic	Good for identical and non-identical data
Standardization of Input Data	International
Data Analysis	Good

Table 4.7: Summary of criteria for relational database.

All relational databases are not platform and application independent data storage applications. For example, MySQL and Oracle are platform and application independent, but Microsoft SQL Server is dependent. Moreover, if a relational database is used to store the parameters of the AMIRIS model, programmers need to spend extra time for maintaining the database system which is cost and time ineffective. All of the database systems are not open-source. The user needs to spend money to buy the license to use them. For instance, MySQL is open-source database system, but Microsoft SQL Server is not open-source. Typed data is an important issue for a software program. The relational database systems provide data as normal text to the software programs. Like Microsoft Access, data are stored with primitive data types in relational database systems but these systems do not provide validation of data when software programs process data. As mentioned earlier, database systems can store data with required data types but it cannot confine input data in a specific range. Moreover, Java programming language does not provide any parsing facility to fetch the input data from relational databases. Like

Microsoft Access, every relational database requires a connection with software program which takes extra time. For this reason, the performance of the software program can be degraded. Different applications can access same data from a database concurrently. To build a database into a database system and manage data into created database, the user should have sufficient knowledge about database systems. So it is difficult to create a database and maintain data for a naive user. Data access from a database system is comparatively easy and simple by software programs. Database system can store identical and non-identical data. Moreover, it can manage large amount of data and supports most of the written languages by human. Database system provides some powerful features to create different reports (i.e. Crystal Report) which are useful to analyze the data.

4.2.5 Extensible Markup Language (XML)

XML stands for Extensible Markup Language which is called markup language. It is derived from SGML (Standard Generalized Markup Language) and defined by W3C (World Wide Web Consortium). It is not a programming language like Java, C++, C Sharp. It is a non-procedural language and grammar for storing data in a structured way. It is a technology of data structuring for formatting the data in documents to use in various applications. XML is a standard to exchange data on the web, with other applications and different platforms. It describes and defines a class of data objects called XML documents. The XML document structure is very easy to understand by human and computer.

The working team of World Wide Web Consortium (W3C) has defined some important design goals of XML. The goals are in the following:

- XML is developed to exchange data over the internet in a straightforward manner.
- Most of the applications and all operating system will support XML.
- Since XML is a subset of SGML, XML would be compatible with SGML.
- XML would be simple for developers so that they can easily write programs to process the XML documents.
- The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
- XML documents should be clearly readable by human and computer.
- It would be easy to prepare XML design.
- The design of XML shall be formal and concise.

- XML documents would be easy to create.
- Terseness in XML markup has minimal importance.

Nowadays, compatible data transfer is a vital and essential issue over the internet and between heterogeneous applications as well as between different operating systems. XML has such some important features that have made it suitable and appropriate data format to transfer the data. The important features are as follow:

- XML is a human and computer-readable format.
- Since XML supports Unicode, it allows almost any kind of information that is written by human languages.
- The data structures of computer science can be denoted by XML. For example, lists, records, and trees etc.
- Since XML is self-describing document format, it describes itself its structure, element names and specific values of these elements.
- XML has the strict syntax and parsing requirements. For these reasons, the necessary parsing algorithms are really simple and efficient to parse the XML documents.
- XML is extensible because new tags can be defined by users as much they required
- Hierarchical data can be modeled to any level of complexity
- Data can be checked to correct the structure of
- XML can validate its data using validation methods.

Currently, XML is used to store and process well-formed data for both online and offline applications as well. The advantages to use XML rather than other storage applications are:

- Since XML follows the international document format, it is a robust, logical and more verifiable document format.
- The hierarchical structure is suitable for most (but not all) types of documents;
- It is like a plain text file and free of encumbrance by licenses or restrictions;
- It is independent for all operating systems and almost all software applications.
- SGML, XML predecessor, has been used since 1986. So XML is an extensive experience and software availability.
- XML is very useful for small amount of data and small operations.
- XML can organize the data in a hierarchical approach.

Figure 4.3 describes document structure of a plain text file. Listing 4.1 represents an

example of XML document. Finally plain text and XML have to be compared to get a better idea about XML.

Figure 4.3: An example of Document Structure of plain text file.

Source: Author's illustration

Listing 4.1: An example of the structure of an XML document.

```
<agents>
        <agent>
                <agentId>1</agentId>
                <agentName>Netzbetreiber</agentName>
        </agent>
        <agent>
        <agentId>11</agentId>
        <agentName>WAB</agentName>
        <kommStromList>
                <kommStrom>
                        <kommId>WAB_11_VK1_PRIVAT_NB</kommId>
                        <agentenklasse>Netzbetreiber</agentenklasse>
                        <active>true</active>
                </kommStrom>
        </kommStromList>
    </agent>
</agents>
```

To better understand the difference between plain text and XML document, the following two questions have an important impact.

– Which one is easier and comprehensive to read?

– Which one is easier and comprehensive to process for software applications?

Figure 4.3 shows that it is very difficult to understand the exact meaning of texts. On the other hand, Listing 4.1 demostrates that all required information in XML are enclosed by meaningful tags. These information are easily human and computer readable. The user can easily extract the name of an agent from this XML document which is surrounded by `<agentName>` and `</agenName>` tags. It is technically known as `<agentName>` element. Moreover, computer applications can simply process this information.

XML document itself is a self-describing data structure. The structure of XML is a tree oriented and XML document can be modeled as ordered and labeled tree. Every XML document must have a document node called root node. Root node is the ancestor of other nodes of the same document.

If our family history is analyzed, we are descendants of our family tree. The ancestors are on the top of family tree as well as latest children are on the bottom of family tree. By using a tree structure, it is much easy and convenient to find parents, grandparents and other relationships of children.

If a programmer follows some fundamental rules, XML tree is very easy to construct. XML document from Listing 4.1 is converted into a tree structure with conventional XML elements to understand these terms of a tree.

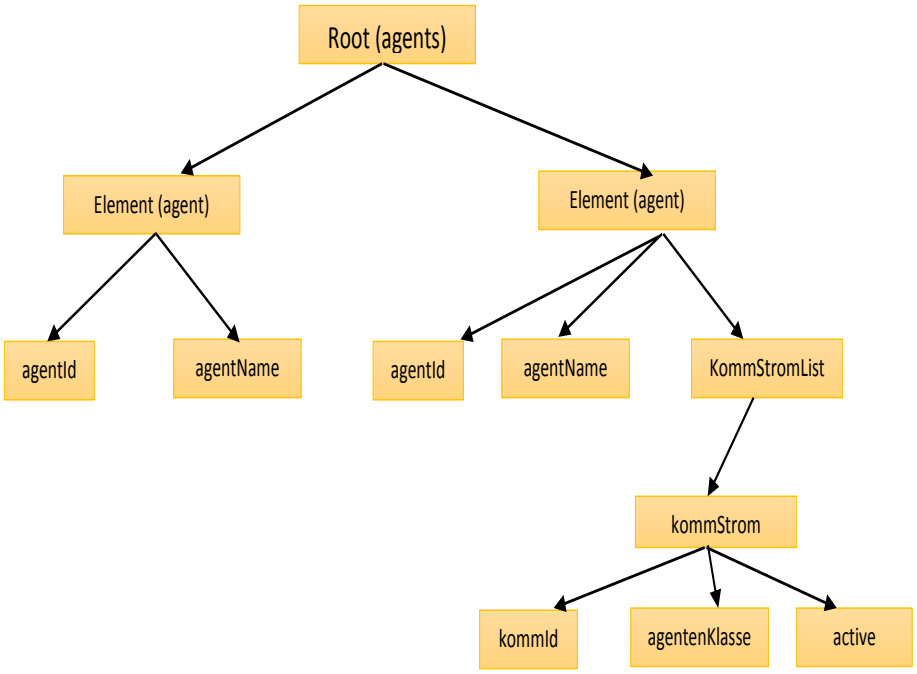

Figure 4.4: An example of Tree Structure of an XML Document (partial).

Source: Author's illustration

Figure 4.4 shows that `agents` is the *root* element of the XML document and `agents` is on the top of XML tree. In other words, all other elements are descendants of `agents` element.

Some important syntaxes to create a structured XML document are:

Tag: A tag is the text between the left angle bracket `(<)` and the right angle bracket `(>)`. There are two kind of tags. They are:

- Opening tag - `<element>` `(i.e. <name>)`
- Closing tag - `</element>` `(i.e. </name>)`

XML tags are case-sensitive. For example, the tag `<Name>` is not same as `<name>`.

Element: An element describes about the data which it contains. An elemenent can also include other elements.

- Section of data beginning with `<tag-name>` and ending with matching `</tag-name>`.
- Content of an element can be text and/or child element(s) or mixture of text and child element(s).

Attributes: An attribute is a property of an element. An element can contain multiple attributes in XML document.

- Attributes can be used to describe elements.
- Attributes are specified by `name="value"` pair inside the starting tag of an element.
- Attribute names must be unique within the element.

Table 4.8 demonstrates the summary of the defined criteria in Table 4.1 for XML as a suitable storage application.

Extensible Markup Language (XML) is platform and application independent storage application. Since it is an open-source application, the user does not require to buy the license to use it. Typed data is very useful to optimize the code and increase the performance of software programs. Software programs can retrieve data with desired data types from XML document. Data validation is a vital issue for most of the software programs. XML supports external validation mechanism to provide the expected data to software programs. Most of the storage applications can specify different data types but they cannot restrict the input data in a certain range. In contrast, XML can confine the input data in a specific range by using facets (restrictions) and regular expressions from external validation mechanism. Repast Simphony with Java has been used to develop the

AMIRIS model. Java programming language provides different parsers to process data from XML document. Moreover, establishment of a connection between a software and a data storage application requires extra time which can decrease the performance of the software. In order to retrieve data from XML document, software programs do not need to setup a connection. Software programs just use the path of XML document where it is located.

Storage Application / Criteria	Extensive Markup Language (XML)
Platform Independency	Independent
Application Independency	Independent
Open-source Facility	Open source
Processed Data by Software Program	Typed data
Validation	Possible
Input Restriction	Possible
Java Parser Facility	Yes
Connection Orientation	No
Concurrency Control Mechanism	Support
Flexibility	Flexible
Human Understandability	Easy
Computer Understandability	Easy
Scalability	Scalable
Identical Characteristic	Good for identical and non-identical data
Standardization of Input Data	International
Data Analysis	Good

Table 4.8: Summary of criteria for Extensive Markup Language (XML).

Like relational database systems, XML provides concurrency control mechanism. This means different software programs can access XML document at the same time. XML document is very easy to create. Furthermore, XML document is structured with meaningful names of elements and attributes. For this reason, XML document is computer and human readable. It is very suitable application for identical and non-identical data, and can store large amount of data. Since XML follows international standard, it supports most of the written languages by human. XML is also useful for statistical analysis and

graphical representation of data. OLAP (Online Analytical Processing) and data mining are used to analyze data from XML document.

4.2.6 Evaluation of Storage Applications

In the past different data storage applications have been analyzed and investigated in detailed. Table 4.9 represents the summary of all of the discussed storage applications with defined criteria to select an appropriate and suitable storage application for storing and managing the parameters of the AMIRIS model.

Storage Application / Criteria	Microsoft Excel	CSV	Microsoft Access	Relational Databases	XML
Platform Independency	Dependent	Independent	Dependent	Not all databases	Independent
Application Independency	Dependent	Independent	Dependent	Not all databases	Independent
Open-source Facility	Not open source	Open source	Not open source	Not all databases	Open source
Processed Data by Software Program	String	String	String	String	Typed data
Validation	Not possible	Not possible	Not possible	Not possible	Possible
Input Restriction	Not possible	Not possible	Not possible	Not possible	Possible
Java Parser Facility	No	No	No	No	Yes
Connection Orientation	No	No	Yes	Yes	No
Concurrency Control Mechanism	Does not support	Does not support	Support	Support	Support
Flexibility	Flexible	Flexible	Inflexible	Inflexible	Flexible
Human Understandability	Easy	Difficult	Difficult	Difficult	Easy
Computer Understandability	Difficult	Difficult	Easy	Easy	Easy
Scalability	Not scalable	Not scalable	Scalable	Scalable	Scalable
Identical Characteristic	Good for identical data	Good for identical data	Good for identical and non-identical data	Good for identical and non-identical data	Good for identical and non-identical data
Standardization of Input Data	International	International	International	International	International
Data Analysis	Good	Not good	Good	Good	Good

Table 4.9: Summary of the defined criteria for all data storage applicaitons.

Selecting a suitable data storage applications with the fulfilment of all requirements for a specific software is really difficult. In order to select an appropriate and suitable data storage application, some important criteria have been defined according to the requirements of the paramerization of the AMIRIS model. Table 4.9 shows that Microsoft Excel is a useful and powerful application to store large amount of data and to provide statistical analysis and graphical representation. But Microsoft Excel does not satisfy most of the defined criteria to choose as an appropriate and suitable data storage application for storing and managing the parameters of the AMIRIS model.

Comma Separated Values (CSV) file is a powerful and convenient application to interchange data between heterogeneous applications and operating systems. Like Microsoft Excel, it does not fulfil most of the criteria for storing and maintaining the parameters of the AMIRIS model.

On the other hand, Microsoft Access provides the concurrent control mechanism and can manage large amount of data. But it cannot meet most of the defined criteria to select as an appropriate data storage application for managing the parameters of the AMIRIS model.

Relational database is a powerful and sophisticated tool for managing large volume of data. Although it fulfils some of the defined criteria as an appropriate data storage application, it cannot satisfy most of the defined criteria.

Table 4.9 demonstrates that Extensible Markup Language (XML) is very useful and appropriate data storage application. It is very easy to create and easy to use into software programs. Moreover, XML satisfies almost all defined criteria to choose as an appropriate and suitable data storage application to store and manage the parameters of the AMIRIS model.

Finally, research and investigation show that Extensible Markup Language (XML) fulfils all of the deliberated criteria. So XML has been selected to store and manage the parameters for defining the model criteria and for initializing different agents of the AMIRIS model.

4.3 Validation Mechanisms for XML

Data Validation is an important concern in software development to optimize the code and to increase the performance of software programs. Different mechanisms are available to validate data in software programs. Different criteria are reflected for different software programs to choose a suitable validation mechanism regarding the software requirements.

Since XML has been selected as data storage application to store the parameters of the AMIRIS model, an appropriate XML validation mechanism has to be chosen for validating the parameters. Several XML validation mechanisms exist to define the structure of XML document and validate XML data such as XML Schema Definition (XSD), Document Type Definition (DTD), RELAX NG and Schematron. Some important criteria are listed in Table 4.10 which are defined for selecting an appropriate XML validation mechanism for validating the parameters of the AMIRIS model.

List of Considered Criteria
Understandability
Object-oriented XML schema language
Data typing support
Namespace support
User-defined data typing
Restrictions (facets)
Reusability on same document
Data Conversion
Required new language learning
Uniqueness and foreign key
Simplicity and Flexibility
Regular expression facility

Table 4.10: List of criteria for the selection of XML validation mechanism.

Understandability: An XML validation mechanism should easily be human and computer readable.

Object-oriented XML schema language: Different XML validation mechanisms are available to define the structure of XML document and to validate XML data. Most of the validation mechanisms do not provide object-oriented facility. They just follow the fundamental rules to define the structure of XML document and validate data. Since the AMIRIS model is developed with Java programming language which is an object-oriented language, the XML validation mechanism should be object-oriented so that programmers

can use the object-oriented features to process the parameters.

Data typing: It is easy and convenient to parse data with expected data type by software programs. If the data storage application provides data without expected data types to software programs, software programs need to convert input data into expected data format which is time consuming and cost ineffective. On the other hand, input data with defined data types reduces the coding complexity and provides the convenient data processing into software programs. So the AMIRIS model requires such a validation system which supports predefined data types for the parameters to optimize the coding structure.

Namespace supporting: Sometimes programmers can incorporate multiple XML schemas for validating an XML document. The main problem to use multiple schemas on the same document is naming collision of the element. This means an element can be defined in different schema with same name which has different meaning of the element for different schemas. For example, there are two XML schemas named as *customer* and *employee*. Both schemas can have same element name like *first_name*. Although the name of the element in both schemas is same but it provides different meanings for each schema. The element name in *customer* schema defines the first name of a customer, whereas in *employee* schema it defines the first name of an employee. To overcome this problem, programmers can use a schema validation mechanism which supports namespace property to avoid naming collision. Since programmers can use multiple XML schemas in same XML document to validate the parameters of the AMIRIS model, an appropriate validation mechanism is required for the AMIRIS model to avoid the naming collision.

User-defined data typing: Primitive data types only provides facility to define input data with desired data types but does not provide any facility to define a segment of an XML schema in a single variable. User-defined data types can define a segment of a schema via a single variable which is very useful to reuse the elements and attributes which are frequently used in same schema document.

Restrictions: Primitive data types can organize input data into different categories (i.e. integer, double, string). But they cannot restrict input data in a specific range. For example, the percentage of total mark of an exam is not less than 0 and not more than 100. Some input data (i.e. share of direct marketing) in AMIRIS model must be restricted in a certain range. So the AMIRIS model requires a validation system which can specify input range.

Reusability: It means a chunk of code can be used to define another module, element or functions. A validation system with this feature is useful and convenient to reduce

extra effort of programmers.

Data Conversion: It defines conversion from one format of encoding data to another format (from string to integer). Sometimes software programs require to convert from one data format to another. Some validation systems provide this feature which optimizes the code of software programs.

Requirement of new language learning: Since XML has been selected as data storage application for the parameters of the AMIRIS model, it is very convenient if the validation system follows same syntaxes of XML document. Thus programmers do not need to learn new schema language to create XML schema.

Uniqueness and foreign key: Most of the relational database systems provide unique key and foreign key properties which are useful to establish a relation between different tables. Some validation systems also have such facilities to make a relation between different elements of XML document. The AMIRIS model needs such validation system to know the contact relationship between different agents.

Simplicity and Flexibility: Simplicity and flexibility of creating validation schema, adding new elements and attributes into schema and altering existing elements and attributes are important issues for selecting a validation system.

Regular expression facility: Regular expressions provide basic and extended standard of the grammar and syntax. A validation system is required for the AMIRIS model which can provide the specific pattern of the parameters.

DTD and XSD are very popular and frequently used to validate XML document. This section describes and evaluates only DTD and XSD according to the criteria which are defined above.

4.3.1 Document Type Definition (DTD)

DTD stands for Document Type Definition which defines the structure of XML document. For example, what elements, attributes are allowed into XML document. DTD is usually used to make sure the conformity of an XML document. It is a syntactical refinement tool to define a well-structured XML document. Moreover, it is used to specify the order and position of elements as well as which elements are allowed inside other elements. DTD also specifies which elements are mandatory and which are optional in XML document. Furthermore, it defines the occurrence of the elements and validates the required attributes of those elements. Although it is an important part of a well-formatted

XML document, it is not mandatory to have a DTD into XML document. DTD is a combination of all tag names and specification of the form. There are different ways to incorporate DTD into XML document. Two ways are mentioned here. One way is that DTD can be embedded into XML document itself. Another one is that DTD can be put into a separate file and the programmer can use the reference to XML document. The main advantage of separate DTD file is to use it to other XML documents as much as the programmer wants. Programmers can use multiple DTD files to a single XML document. Incorporating multiple DTD files can be cause of naming collision. If two or more DTD files define a tag with the same name, that tag can have naming collision. In order to identify each DTD separately, *namespace* is used in XML document. For example, if two DTD files include a definition of an element with the tag name *agentname*, one DTD could be declared in the namespace *producer* and another in the namespace *buyer*. Then there are two separate tag names *producer:agentname* and *buyer:agentname* available to use in XML document.

This section illustrates the embedding processes of DTD file into XML document, describes element and attribute declaration of DTD. Finally this section summaries the criteria which are defined above to select DTD as a validation mechanism for validating the parameters of the AMIRIS model.

Single File and Multiple Files

The `DOCTYPE` keyword is used to insert DTD information into the XML document and *standalone* keyword is either set to *yes* or *no*. If *standalone* is set to *yes*, it indicates that XML contents and DTD information are in a single file and parser will handle out only a single file. On the other hand, if *standalone* is to *no*, it indicates that XML and DTD are in separate files and parser will be ready to handle multiple files even though DTD is embedded into XML document.

`DOCTYPE` keyword is used to define the DTD information which is shown in Listing 4.2 and Listing 4.3. Listing 4.2 demonstrates DTD information with single file and Listing 4.3 illustrates DTD information with separate files. `DOCTYPE` keyword contains an opening and a closing brackets. Everything inside those two brackets is part of DTD. An element is required for `DOCTYPE` keyword itself. This element is known as *root* element. Listing 4.2 shows that *agents* is the root element of DTD and XML and there are in total 9 `ELEMENT` declarations. The declaration of an `ELEMENT` defines the necessary contents inside the `ELEMENT`. The root `ELEMENT` *agents* contains only *agent* element with an asterisk mark which indicates that the *agents* element can have zero or several *agent* elements.

Listing 4.2: An example of XML document with internal DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE agents[
<!ELEMENT agents (agent)*>
<!ELEMENT agent(agentId, agentName, (kommStromList)*)>
<!ELEMENT agentId (#PCDATA)>
<!ELEMENT agentName (#PCDATA)>
<!ELEMENT kommStromList (kommStrom)*>
<!ELEMENT kommStrom (kommId, agentenklasse, active)
<!ELEMENT kommId (#PCDATA)>
<!ELEMENT agentenkasse (#PCDATA)>
<!ELEMENT active (#PCDATA)>
]>

<agents>
        <agent>
                <agentId>1</agentId>
                <agentName>Netzbetreiber</agentName>
        </agent>
        <agent>
        <agentId>11</agentId>
        <agentName>WAB</agentName>
        <kommStromList>
                <kommStrom>
                        <kommId>WAB_11_VK1_PRIVAT_NB</kommId>
                        <agentenklasse>Netzbetreiber</agentenklasse>
                        <active>true</active>
                </kommStrom>
        </kommStromList>
    </agent>
</agents>
```

The *agent* element must contain two elements (*agentId, agentName*) and may contain an element (*kommStromList*). The *agentId* and *agentName* keywords must be appeared exactly once inside the *agent* element because of default modifier. On the other hand, *kommStromList* element can be occurred zero or more times in the *agent* element because of asterisk modifier. The elements must be appeared sequentially because of comma

separated declarations. The *agentId, agentName* elements have to contain only `PCDATA` (Parse Character Data), which means they can comprise only a sequence of characters, *kommStromList* element will contain other elements. DTD can define entities for the shortcut of an element or special character or a sequence of characters.

Listing 4.3: An example of XML document with external DTD.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE agents SYSTEM "../dtd/agents.dtd">
3
4  <agents>
5          <agent>
6                  <agentId>1</agentId>
7                  <agentName>Netzbetreiber</agentName>
8          </agent>
9          <agent>
10         <agentId>11</agentId>
11         <agentName>WAB</agentName>
12         <kommStromList>
13                 <kommStrom>
14                         <kommId>WAB_11_VK1_PRIVAT_NB</kommId>
15                         <agentenklasse>Netzbetreiber</agentenklasse>
16                         <active>true</active>
17                 </kommStrom>
18         </kommStromList>
19     </agent>
20 </agents>
```

Listing 4.3 shows that `DOCTYPE` keyword declaration for separate DTD files is almost same like single file. In addition it has an extra keyword which is called `SYSTEM`. If the programmers want to use DTD information in a separate file, `SYSTEM` keyword is required to locate the external DTD file. There are two types of location paths of a file: *relative* and *absolute* path. The relative path has been used in this example.

ELEMENT Declarations and Definitions in DTD

`ELEMENT` declarations with basic syntax are really easy and simple in DTD. The `ELEMENT` keyword is used to define the elements. It defines the form of tag name of an element

and specifies which contents will be contained inside that element. Some basic ways to specify an element:

No Element: If an `ELEMENT` is declared with `EMPTY` keyword, it does not contain any data inside the element. It is basically used for special tags. For example, `<br>` tag is used for a line break.

Simple Content: When an `ELEMENT` is declared with `#PCDATA or #CDATA` (known as character data), it contains only *parse character data* or *character data*. Normally the programmers use `#PCDATA` keyword which is easy to parse by the parsers.

Element Declaration with Children: When an element is declared as a parent element, it contains a list of other elements as its children. For example, `<!ELEMENT agents (agent)*>`. Here *agents* is the parent element and *agent* is the child element of *agents*.

Unrestricted Content: If an element is declared with `ANY` keyword, there is no restriction about data. That means, it can have any information either list of other elements or a sequence of character data.

Mixed Content in an Element: The element can contain both text and other elements. If the programmers require the combination of above four methods, it is very useful in that case. A simple example of mixed content is below: `<!ELEMENT element-name (#PCDATA | element-name1 | element-name2 | element-name3)>` where `pipe (|)` is used to make a choice list. That means, the programmer can use either `#PCDATA` or combination of all elements.

Declaration of Attributes

DTD uses `ATTLIST` keyword to declare the list of attributes for a specific element. This list of attributes is a part of that element. There are some keywords (i.e. `#IMPLIED, #REQUIRED, #FIXED`) in DTD which are used to specify the occurrence of the attributes to the element. Examples of attributes declaration are as follows:

```
<!ELEMENT agent (agentId, agentName, KommStromList, stratSEContract)>
<!ATTLIST agent agentType CDATA #REQUIRED>
<!ATTLIST agent riskType CDATA #IMPLIED>
```

Another way is to define attributes.

```
<!ELEMENT agent (agentId, agentName, KommStromList, stratSEContract)>
<!ATTLIST agent agentType CDATA #REQUIRED
                riskType CDATA #IMPLIED>
```

If `#REQUIRED` keyword is applied to an attribute, this attribute must be appeared inside the element. For `#IMPLIED` keyword the attribute may be either appeared or not. In case of `#FIXED` keyword the attribute will be appeared with a fixed quoted string in the element.

Declaration of ENTITY keyword

Basically `ENTITY` keyword is used to define a shortcut for a special character or a sequence of characters or a whole element with its child elements. It can be specified in the following:

```
<!ENTITY agent_name "Wind Anlage Betreiber">
```

The programmer can use it into XML document by the following way.

```
<agentName>&agent_name</agentName>
```

The value of `agent_name` will be automatically recognized when the programmer uses it.

Occurrence Operators	Descriptions of Operators
Default Operator	If there is no occurrence operator defined in the element, that element must be appeared exactly once.
Plus sign (+)	If plus (+) operator is used in the element, that element must appeared at least once or might be repeated more than once.
Asterisk sign (*)	Asterisk (*) sign indicates that the element can be occurred zero or more times.
Question sign (?)	Question (?) sign indicates that the element can be omitted or appeared at most once.

Table 4.11: The Operators of Occurrence

Source: Brett D. McLaughlin and Justin Edelson (2006)

DTD is a useful and convenient tool to define the structure of XML document and to validate data into XML document. Some important facilities are mentioned below.

- DTD is really easy to write.
- DTD needs few lines of code to create.
- DTD validates the structure of XML document.
- It is very easy to understand the structure of XML document by looking on DTD.

- DTD can easily manage the order and occurrence of the elements.
- DTD is used to insert additional constraints and rules to XML document.
- By using DTD, the parsers can simply identify the errors of XML.

Though DTD has several advantages, it has also some limitations. Few of them are described in the following.

- DTD represents all values as string.
- DTD does not support data types (i.e. integer, float, double)
- DTD cannot restrict the element occurrence into certain limit. For example, the occurrence of an element must be between 5 and 100.
- DTD cannot specify user-defined types.
- DTD does not follow the XML syntax.
- DTD does not follow the object-oriented mechanism. For example, it does not have inheritance.

Table 4.12 presents the summary of the defined criteria in Table 4.10 for DTD as a suitable XML validation mechanism.

Document Type Definition (DTD) is simple and easy to understand by human and computer for both small and large schema. Although object-orientation is important to develop a robust software program, DTD does not provide object-oriented facility to define the structure of XML document and to validate input data. Data typing is an important issue to process expected input data by software programs. Unfortunately, DTD does not offer primitive data types. It specifies the values of elements and attributes with only string. DTD affords namespace feature to avoid naming collision of elements with same name in different schema files. Sometimes user-defined data types are very convenient to use a segment of schema code into another part of schema. But DTD does not have this facility. Some software programs needs to restrict some input data in a specific range but DTD cannot restrict input data. Reusing a chunk of schema code can increase the productivity of software programs. DTD does not have the facility to use a segment of code into the same schema. Sometimes software programs require to convert input data from one format to another for getting expected output. DTD cannot convert input data. Since the structure of DTD is totally different, programmers need to learn new language to create schema and validate input data (see Listing 4.2). DTD cannot relate different elements together and cannot facilitate unique key and foreign key features. Although it is easy to understand and easy to create simple schema, it is difficult to create large

schema using DTD. Moreover, DTD cannot define the pattern of input data.

Validation Mechanism / Criteria	Document Type Definition (DTD)
Understandability	Easy to understand
Object-oriented XML schema language	No
Data typing support	No
Namespace support	Yes
User-defined data typing	No
Restriction (facets)	No
Reusability on same document	No
Data Conversion	No
Required new language learning	Yes
Uniqueness and foreign key	Does not support
Simplicity and Flexibility	Good for simple schema
Regular expression facility	No

Table 4.12: Summary of criteria for Document Type Definition (DTD).

4.3.2 XML Schema Definition (XSD)

The short for XML Schema Definition, XSD is an alternative of DTD. XSD is also called XML Schema. XML Schema is a recommendation of W3C (World Wide Web Consortium) to specify the structure of XML document and validate data into XML document. It is an XML-based schema which is close to the understanding of relational database. Like DTD, XML Schema defines elements and attributes which can appear into XML document, specifies which elements are child elements and the order of the child elements. It also defines the number of child elements inside a specific element. Moreover, XML Schema provides more facilities than DTD. For example, it has object-oriented and data typing features to provide inheritance and constraining of data in the XML document respectively. It provides user-defined data type which is very useful to reuse into another elements and another XML Schema. XML document cannot directly embed XML Schema into the document. XML Schema is created in separate file and the

reference of the schema is used to XML document. Listing 4.4 represents an example of XML Schema and Listing 4.5 shows an example how XML Schema is incorporated into XML document to define the structure of XML document and validate XML data.

Listing 4.4: An example snippet of XSD.

```
<xsd:element name="agents">
<xsd:complexType>
<xsd:sequence>
        <xsd:element name="agent" minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType>
        <xsd:sequence>
                <xsd:element name="agentId" type="xsd:int" minOccurs="1" maxOccurs="1" />
                <xsd:element name="agentName" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                <xsd:element name="kommStrom" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
                                <xsd:sequence>
                                        <xsd:element name="kommId" type="xsd:string" />
                                        <xsd:element name="agentenklasse" type="xsd:string" />
                                        <xsd:element name="active" type="xsd:boolean" />
                                </xsd:sequence>
                        </xsd:complexType>
                </xsd:element>
        </xsd:sequence>
        </xsd:complexType>
        </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

`agents` is the root element in this schema document and contains only `agent` element. The `minOccurs` property of `agent` element is set to 1 (one) which indicates that `agents` must contain at least one `agent` element in XML document. Moreover, `maxOccurs` property is set to "unbound" that indicates `agents` element can have several `agent` elements. On the other hand, `agent` element must contain `agentId` and `agentName` elements once when it appears in XML document. In addition, `kommStrom` element is not mandatory to occur with `agent` element since `kommStrom` element sets 0 (zero) to `minOccurs` property. Since data types for `agentId` and `agentName` are `integer` and `string` respectively, the values must be `integer` and `string` type for `agentId` and `agentName` elements accordingly.

Listing 4.5: An example of XML document with XML Schema reference.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <agents xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:↩
       noNamespaceSchemaLocation="../xsd/agents.xsd">
3      <agent>
4              <agentId>1</agentId>
5              <agentName>Netzbetreiber</agentName>
6      </agent>
7      <agent>
8              <agentId>11</agentId>
9              <agentName>WAB</agentName>
10             <kommStrom>
11                 <kommId>WAB_11_VK1_PRIVAT_NB</kommId>
12                 <agentenklasse>Netzbetreiber</agentenklasse>
13                 <active>true</active>
14             </kommStrom>
15     </agent>
16 </agents>
```

Listing 4.5 demonstrates how the reference of XML Schema is incorporated into XML document. The property of XML Schema `noNamespaceSchemaLocation` is used to include the XSD file into XML document. In this example *relative path* has been used.

Element Declarations in XSD

Since element and attribute are crucial to create XML Schema, only element and attribute declarations are described here. Element declarations of XSD with basic syntax are comparably difficult than that of DTD. The `element` keyword is used to define the elements. Moreover, it specifies the form of the tag-name of an element and defines which elements are inside that element. An element can be defined in different ways. Some of them are mentioned below.

Empty Element: In order to declare an empty element, the element must not have any child element and not have any data type of that element. For example,

```
<xs:element name="agent">
    <xs:complexType>
        <xs:attribute name="agentId" type="xs:positiveInteger"/>
```

```
    </xs:complexType>
</xs:element>
```

Element with children: When an element is defined as a parent element, it includes a list of child elements. For example,

```
<xs:element name="agent">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="agentId" type="xs:positiveInteger"/>
            <xs:element name="agentName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Simple Element: One of the ways to declare an element as a simple element is following.

```
<xs:element name="agentId" type="xs:positiveInteger"/>
```

Attribute Declarations in XSD

The `attribute` keyword is used to define an attribute. Examples of attribute declarations inside an element are in the following.

```
<xs:element name="agent">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="agentId" type="xs:positveInteger"/>
            <xs:element name="agentName" type="xs:string"/>
        </xs:sequence>
         <xs:attribute name="agentType" type="xs:string"/>
         <xs:attribute name="riskType" type="xs:positiveInteger"/>
    </xs:complexType>
</xs:element>
```

XML Schema is a powerful and convenient XML validation mechanism. It provides several advantages to define the structure of XML document and validate input data into XML document.

– XML Schema is an object-oriented XML schema language. For instance, inheritance of element, attribute and data type definitions.

- It supports different data types. For example, integer, float, double etc.
- It supports user-defined data types.
- Reusability of one schema into another schema.
- Restrictions for the occurrence of element and the value of element. For example, the number of occurrence of an element must be between 1 and 100 as well as the value of a specific element must be between 10 and 50.
- Addition of new element to the existing schema is easier.
- It provides data conversion facility.
- Since XML Schema is similar with XML, the programmers do not require to learn new language.
- It is easy to use reference of a schema in the same schema document.
- XML Schema provides the features of regular expression.
- It supports namespace recommendation.
- XML Schema provides uniqueness and foreign key constraints.

Although XML Schema provides lot of advantages to make sure the conformity of XML document, it has also some disadvantages. Some of them are mentioned in the following.

- Sometimes XML Schema can be difficult to understand in case of large schema.
- XML Schema is verbose.
- XML Schema is not possible to write inline, must be defined in a separate file.
- XML Schema requires more lines of code than DTD.

Table 4.13 represents the summary of the defined criteria in Table 4.10 for XSD as a suitable XML validation mechanism.

Sometimes it can be difficult to understand the structure of XSD in case of large schema. XSD provides object-oriented features which is useful to develop a robust software. It offers different primitive data types to process input data with desired data format by software programs. Moreover, it provides namespace feature to avoid naming collision of an element with same name in different schema files. It also offers user-defined data type facility to reuse a segment of code into another part of the schema. XSD can restrict input data in a certain range using different facets.

Validation Mechanism / Criteria	XML Schema Definition (XSD)
Understandability	Sometimes difficult to understand in case of large schema
Object-oriented XML schema language	Yes
Data typing support	Yes
Namespace support	Yes
User-defined data typing	Yes
Restriction (facets)	Yes
Reusability on same document	Yes
Data Conversion	Yes
Required new language learning	No
Uniqueness and foreign key	Support
Simplicity and Flexibility	Good for complex schema
Regular expression facility	Yes

Table 4.13: Summary of criteria for XML Schema Validator (XSD).

By reusing the segments of schema code into another part of the schema XSD helps to increase the productivity of software programs. Furthermore, it has the facility to convert input data from one data type to another. Since XSD follows the syntax and fundamental rules of XML, programmers do not need to learn new language which saves the programmers' time. In addition, XSD can establish relations between different elements using foreign key and can store unique value for the same element with multiple occurrence using unique key. It is very powerful and convenient tool for complex schema and provides regular expressions to define the pattern of input data.

4.3.3 Evaluation of XML Validation Mechanisms

Previously different XML validation mechanisms have been described and analyzed in detailed. Table 4.14 summaries DTD and XSD validation mechanisms with defined criteria to select an appropriate and suitable XML validation mechanism for validating the parameters of the AMIRIS model.

Validation Mechanisms / Criteria	DTD	XSD
Understandability	Easy to understand	Difficult to understand for large schema
Object-oriented XML schema language	No	Yes
Data typing support	No	Yes
Namespace support	No	Yes
User-defined data typing	No	Yes
Restriction (facets)	No	Yes
Reusability on same document	No	Yes
Data Conversion	No	Yes
Required new language learning	Yes	No
Uniqueness and foreign key	Does not support	Support
Simplicity and Flexibility	Good for simple schema	Good for complex schema
Regular expression facility	No	Yes

Table 4.14: Summary of criteria for all XML validation mechanisms.

It is really very tough to choose an appropriate XML validation mechanism which fulfils all requirements of a software program. Every mechanism has some pitfalls. In order to select an appropriate and suitable validation mechanism, programmers should consider some important criteria according to the requirements of the software program. Table 4.14 shows that Document Type Definition (DTD) is easy to understand the structure of the schema. But DTD does not meet all other defined criteria which are required for validating the parameters of the AMIRIS model.

On the other hand, table 4.14 shows that XML Schema Definition (XSD) is sometimes difficult to understand the structure of the schema in case of large schema. But XSD can overcome this problem using modular feature of the schema. Moreover, XSD is able to fulfil all other defined criteria for selecting an appropriate and suitable XML validation mechanism for validating the parameters of the AMIRIS model.

Finally, research and investigation demonstrate that XML Schema Definition (XSD)

meets most of the deliberated criteria. So XSD has been selected to validate the parameters for defining the model criteria and for initializing different agents of the AMIRIS model.

4.4 Parser Mechanisms for XML

Processing, manipulating and creating of XML documents are the key terminologies to use the meaningful and structured XML data in software programs. A parser is a computer program which validates the grammatical structure of the inputs and transfers the parsed data to software applications for processing the input data. Different XML parser APIs (Application Programming Interface) are used in different software programs in order to process and manipulate XML documents according to their requirements. Some useful and popular XML parsers are Document Object Model (DOM), Simple API for XML (SAX), Streaming API for XML (StAX) and Java Architecture for XML Binding (JAXB) which are frequently used in software programs. Programmers consider different criteria for selecting a suitable XML parser regarding the software requirements. In order to select an appropriate XML parser for the AMIRIS model, some important criteria are listed in Table 4.15.

List of Considered Criteria
Simplicity
Flexibility in Data Access
Memory Efficiency
Efficient Tree Structure
Marshalling and Unmarshalling
Validation Assurance
Portability
Customization
Object Relational Model Facility
Increased Productivity
Performance and Efficiency
Flexibility for Non-XML Programmers
Validation Support on Demand

Table 4.15: List of criteria for the selection of XML parser

Simplicity: The XML parser mechanism should be simple and user-friendly for creating, processing, manipulating and maintenance of XML-enabled applications for the programmers.

Flexibility in Data Access: Different XML parsers structure XML data in different ways. Some parsers allow to access XML data in sequential order, some parsers provide the access of XML data in non-sequential order. The selected XML parser for the AMIRIS model should have both data access features so that the AMIRIS model can use both features. That means, the sequential order can be used to maintain the hierarchy among different associated components of agents, and the non-sequential can be used for fast and random access of components of the agents in the AMIRIS model.

Memory Efficiency: Memory efficiency is an important issue to prevent the memory overflow for any software program. The efficient memory management impacts the performance of the software programs significantly. The AMIRIS model needs to execute several times in a single run to get more data for better statistical analysis. Because of several executions of the model it might be cause of memory overflow. So the AMIRIS model requires an XML parser with memory optimization features.

Efficient Tree Structure: The tree structure of XML data is a data representation with hierarchy. It is used to find the relationships among different data elements. In order to identify the associated components with different agents, the AMIRIS model needs a parser which can create tree structure of the XML document with minimal memory requirement.

Marshalling and Unmarshalling Flexibility: Marshalling is the process to generate XML document from class object(s). Node by node processing of XML elements is sometimes difficult and error-prone of the complex XML document. On the other hand, the class objects are very simple and efficient to process and manipulate by the software programs. Unmarshalling is the process for creating class object(s) from XML document. So an XML parser is important for the AMIRIS model to store the agents with their properties in XML document as well as to read the agents from XML document to initialize the agents and add them to the environment of the AMIRIS model.

Validation Assurance: Validation of data types is an important part to optimize the code structure and provide desired data for the software programs.

Portability: Portability means that the software programs implement XML parser in such a way that the components of XML parser can be reused to another software program without making significant changes to the source code. Systems Analysis and Technology

Assessment department, German Aerospace Center has developed different simulation models (i.e. REMix) to analyze the renewable energy. In future the implemented parser in AMIRIS model can also be used for parsing the parameters of REMix model.

Customization: It is a way to customize the existing schema components to object representations in the software programs. Sometimes the dynamic and sophisticated software programs need to change the binding of schema component. Since the AMIRIS model is a dynamic agent-based simulation model, it might be required to customize the schema components.

Object Relational Model Facility: Object Relational Model (ORM) is very efficient and powerful mechanism to process and manipulate the data from relational database as class objects. Most of the XML parsers read the contents from XML document using the element names which are defined in the XML document. If the name of any element is changed in the document, programmers must synchronize the source code statically with the modified element name to fetch the value of that element. Otherwise, the program has to throw an error. On the other hand, some XML parsers provide ORM facility to read XML data as class objects with appropriate getter and setter method for each element and for each attribute. Since these parsers use the getter methods (do not require the element names or attribute names), programmers do not need to make any change in the source code to retrieve the values of the elements and attributes.

Increased Productivity: The complex parsing code is always time consuming and burden for the programmers. The efficient and productive XML parser reduces the implementation time and saves the extra cost of the software programs.

Performance and Efficiency: It means that how fast and efficiently an XML parser processes XML data in the software program. A parser can be fast and efficient for small XML document and another can be for large document. Although currently the amount of the input parameters in AMIRIS model is small, but it will be increased in the future. For this reason, the AMIRIS model needs such an XML parser which is suitable for both small and large amount of data.

Flexibility for Non-XML Programmers: The programmers require sufficient knowledge about the structure and processing mechanisms of XML document to process data in case of most XML parsers. An XML parser is required for the AMIRIS model so that the non-xml programmers can easily work with this parser.

Validation Support on Demand: When working with a content tree corresponding to an XML document, it is often necessary to validate the tree against the constraints in

the source schema. The AMIRIS model needs such an XML parser to get accurate and error-free results.

This section introduces and evaluates different parser mechanisms according to the above defined criteria. Since Repast Simphony with Java has been used to develop the AMIRIS model, this section describes only Java methodologies to process XML document using different parsers.

4.4.1 Document Object Model (DOM)

DOM stands for Document Object Model which is a standard Application Programming Interface (API). It is a cross-platform and language-independent mechanism which is defined by W3C (World Wide Web Consortium). It is known as tree-based model. It defines the logical structure of XML document as well as processes and manipulates XML document. Since DOM stores data as a tree structure in memory, it can easily traverse the elements of XML document. DOM is also used to build documents, and add, update or delete elements and contents.

Figure 4.5 demonstrates the parsing mechanism of XML document in DOM parser. DOM parses the entire XML document at a time. The document is stored into memory as a tree structure. The software programs then process the tree to fetch the required data. DOM parser creates XML document using serialization mechanism.

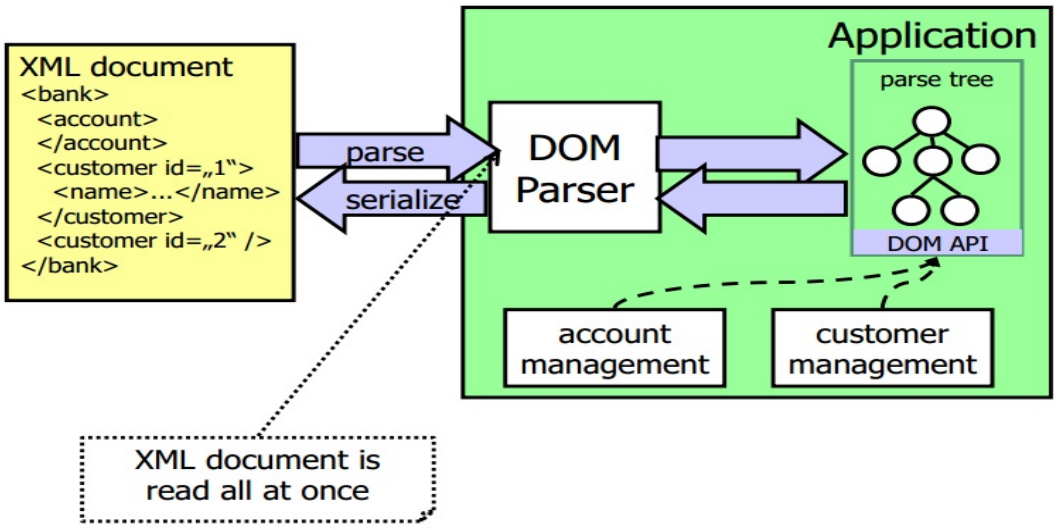

Figure 4.5: DOM API Architecture.

Source: Net-based applications lecture, Uni-Stuttgart

Figure 4.6 shows the graphical representation of the parse tree in DOM parser which is taken from Listing 4.3.

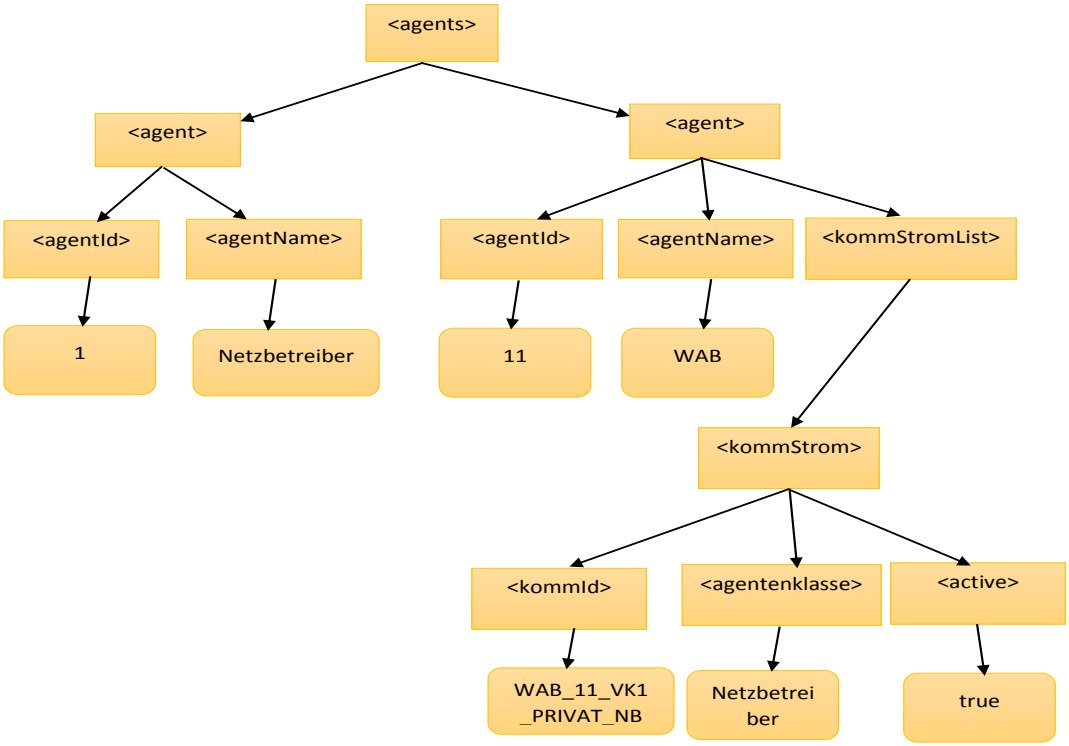

Figure 4.6: Graphical Representation of DOM Parser.

Source: Author's illustration

DOM parser can implement an XML document in both ways: *tree* and *structured objects*. So DOM is called logical object model which may be implemented in any appropriate way. Figure 4.6 shows that DOM creates the logical structure of XML document as a tree. Every XML document can have several element nodes, zero or more processing instructions and comments. Each document must contain a *root* element which is also known as *document* element and may enclose zero or one *docType* element to define validation of the XML document. In the graphical representation of DOM the nodes do not represent a data structure. Each node represents an object and it has own function and identity. The logical object model DOM identifies the following important features:

- The interfaces and objects used to represent and manipulate a document
- The semantics of these interfaces and objects - including both behavior and attributes
- The relationships and collaborations among these interfaces and objects

Extracting and Reading XML Document using DOM

A DOM parser stores the entire XML document into a memory-resident tree structure. All information of an XML document are contained in the tree nodes. The nodes are interconnected with parent/child relationships (Figure 4.6). The application programs can access the nodes to fetch their desired information. Listing 4.6 shows an example of DOM implementation how to read an XML document, how to check errors during document reading and how to access nodes to fetch the parsed data. DOM parser is also used to create and modify the XML documents.

Listing 4.6: Processing to parse data from XML document using DOM parser.

```
1  public class DOMParser
2  {
3      SchemaFactory schemaFactory = SchemaFactory.newInstance(XMLConstants.↩
           W3C_XML_SCHEMA_NS_URI);
4      Schema schema = schemaFactory.newSchema(new File("agent_xsd"));
5      DocumentBuilderFactory parserFactory = DocumentBuilderFactory.newInstance();
6      parserFactory.setNamespaceAware(true);
7      parserFactory.setValidating(true);
8      parserFactory.setSchema(schema);
9      DocumentBuilder parser = parserFactory.newDocumentBuilder();
10     Document doc = parser.parse(new File("agents.xml"));
11     //Iterating through the nodes and extracting the data.
12     NodeList agentList = doc.getElementsByTagName("agent");
13     for (int i = 0; i < agentList.getLength(); i++) {
14         Node agentNode = agentList.item(i);
15         if (agentNode.getNodeType() == Node.ELEMENT_NODE) {
16             Element agentElement = (Element) agentNode;
17             System.out.println("Agent ID : " + agentElement.getElementsByTagName("↩
                   agentId").item(0).getTextContent());
18             System.out.println("Agent Name : " + agentElement.getElementsByTagName("↩
                   agentName").item(0).getTextContent());
19         }
20     }
21 }
```

The static method `SchemaFactory.newInstance()` is used to instantiate the `SchemaFactory` object. The `Schema` instance is created by invoking `SchemaFactory.newSchema()` method. The `newSchema()` method can contain any validation schema (i.e. DTD, XML Schema) as an argument. This schema object is used to define the structure of XML document and fetch appropriate data from XML document.

`DocumentBuilderFactory` uses `DocumentBuilderFactory.newInstance()` static method

to create its new object. This object has three options: `setvalidating()`, `setNamespaceAware()` and `setSchema()`. First two methods have a boolean argument. The `setValidating()` method with a *true* argument indicates that the DOM parser must validate the XML document. The `setNamespaceAware()` with a *true* argument specifies that the DOM parser will process the namespace names of validation mechanism (i.e. DTD or XML Schema) and the element tag names. Finally, the `setSchema()` method with a *schema* argument is invoked to `DocumentBuilderFactory` to parse the XML document properly. The `DocumentBuilder` object is mainly DOM parser. In order to instantiate, `DocumentBuilder` calls the `newDocumentBuilder()` method which is invoked with `DocumentBuilderFactory` object.

`Document` object is created by `DocumentBuilder` instance and `parse()` method parses the XML document to use data in the software applications. The `Document` object is the root of the tree from the resulting parsed XML document. The `Document` instance is an interface that provides several useful methods to access data from the tree. The `Node` interface is an essential part of `Document` interface. It specifies the methods to identify the node types, find the location of nodes and get the relationships among nodes. `NodeList` is another interface which contains a set of identical `Node` objects. A `Node` object is also known as `Element` object. In this example, the `getAttribute()` method is used to retrieve the attribute value of a particular node or element. This method contains a string-type argument which specifies the name of the attribute. The `getElementsByTagName()` method fetches all sub elements and `item()` method provides the positions of the sub elements. Lastly, `getTextContent()` function retrieves the text content of a certain sub element.

Table 4.16 summarizes the deliberated criteria which are listed in Table 4.15 for Document Object Model as a suitable XML parser.

XML parser should be efficient and suitable to process any type (i.e. small, large, simple and complex) of XML document. DOM is an efficient and suitable for small and simple XML documents, whereas the processing of large and complex XML document is difficult. DOM parser requires more memory to process the large document even though the structure of the document is simple. Moreover, in the complex document the elements are deeply nested. For this reason, it is difficult to write a program which can traverse every element of the document. Since DOM parser provide the facility to efficiently make a parse tree from XML document. So DOM API provides the flexibility to access data from XML document and it can traverse data in both sequential and non-sequential manner. It stores the entire XML document into a memory-resident tree structure which is required more memory. So DOM is not the memory efficient parser. Sometimes the

software applications not only need to parse (unmarshalling) the XML document but also require to create (marshalling) XML document from class object (s) for further use. DOM parser offers the XML document creation facility, though it is difficult. It can parse both validated and non-validated XML document. DOM can assure the validation in case of parsing from XML document, while it cannot validate when it creates XML document. It has portability feature. This means the components of DOM parser can be reused into different software applications without making significant changes.

XML Parser / Criteria	Document Object Model (DOM)
Simplicity	Simple for small document
Flexibility in Data Access	Flexible
Memory Efficiency	Inefficient
Efficient Tree Structure	Yes
Marshalling and Unmarshalling	Yes
Validation Assurance	No
Portability	Yes
Customization	No
Object Relational Model Facility	No
Increased Productivity	Yes
Performance and Efficiency	Not good
Flexibility for Non-XML Programmers	Inflexible
Validation Support on Demand	No

Table 4.16: Summary of criteria for DOM parser

DOM parser can easily customize the XML document but cannot customize the existing schema components. Object Relational Model (ORM) is very efficient and powerful mechanism to process and manipulate data as class objects from relational database. This mechanism is also possible for XML document. But DOM parser does not have this facility. In order to process and manipulate XML document in DOM, programmers require more lines of code which is time consuming (see Listing 4.6). DOM parser is a slower process because of more memory usage and traversing data from one tree node to another. It is very easy and simple for small document and also very efficient in case of existing XML document modification. It is not flexible for non-XML programmers because the programmers should have sufficient knowledge on the structure and processing mechanisms of XML document for accessing and manipulating the documents (see Listing 4.6). As mentioned earlier, DOM parser cannot modify the existing validation

schema. So validation is not possible on runtime.

4.4.2 Simple API for XML (SAX)

The short for Simple API for XML, SAX is an event-based push parser API for XML documents which is developed by XML-DEV mailing list. It is a combination of Java methods, interfaces and classes which are used to read XML document and break the document information into small pieces of the inputs which are delivered to the software applications. Since the processing of an element does not depend on the previous or next element, so it is called *state independent* processing of XML document. SAX parser generates a stream of *events* from XML document rather than a tree structure. For this reason, it is called *event-based* parser.

Figure 4.7 demonstrates the parsing mechanism of XML document in SAX API.

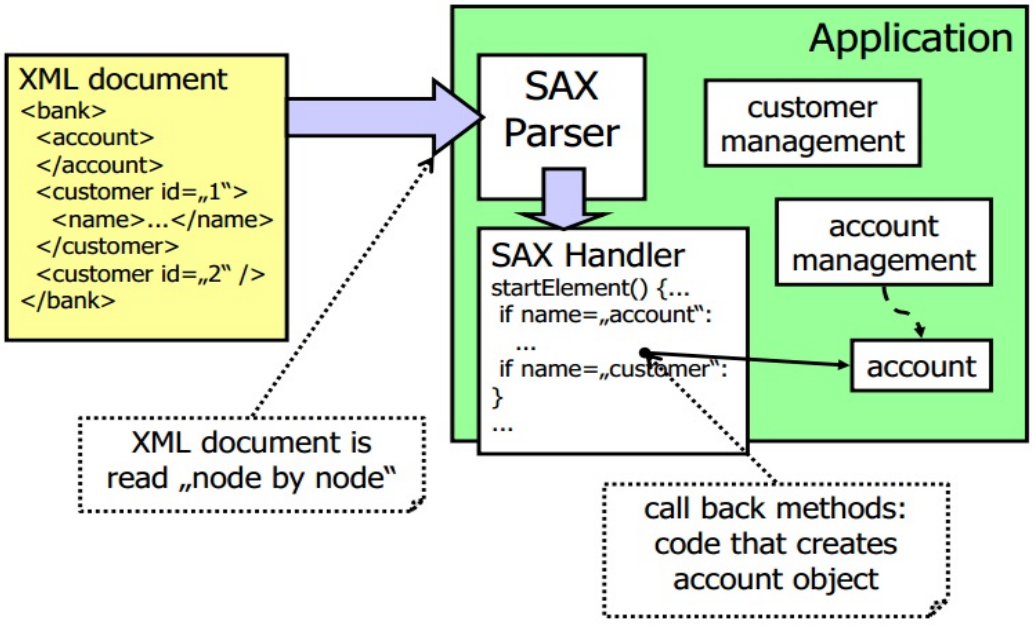

Figure 4.7: SAX API Architecture.

Source: Net-based applications lecture, Uni-Stuttgart

SAX parser reads XML document "node by node". Each node represents an event. SAX Handler class is implemented to process the events. There are several types of events in SAX such as `startElement, endElement, startDocument, endDocument`. Programmers implement their necessary event methods to get the information which is required. If the name of `startElement` event is "account", SAX Handler class will invoke *account* instance to process the parsed data. On the other hand, if the name of `startElement`

event is "customer", SAX Handler class will call *customer* instance to process the parsed data.

SAX parser is a straight approach to read XML document. It parses the data in a sequential manner and delivers the parsed data to the software applications. Listing 4.7 shows an example how to parse and process an XML document (agents.xml) in SAX parser.

Listing 4.7: Processing to parse data from XML document using SAX parser.

```
1  public class SAXParserExp {
2
3      SchemaFactory schemaFactory = SchemaFactory.newInstance(XMLConstants.↩
           W3C_XML_SCHEMA_NS_URI);
4      Schema schema = schemaFactory.newSchema(new File("agent_xsd"));
5      SAXParserFactory parserFactory = SAXParserFactory.newInstance();
6      parserFactory.setNamespaceAware(true);
7      parserFactory.setValidating(true);
8      parserFactory.setSchema(schema);
9      SAXParser parser = parserFactor.newSAXParser();
10     DefaultHandler handler = new SAXHandler();
11     parser.parse(new File("agents.xml"), handler);
12 }
13
14 class SAXHandler extends DefaultHandler {
15
16     public void startElement(String localName) {
17         if (localName.equals("agent")) System.out.println("Start of an agent element↩
               ");
18     }
19     public void endElement(String localName) {
20         if (localName.equals("agent")) System.out.println("End of an agent element")↩
               ;
21     }
22     public void endDocument() {
23         System.out.println("End document");
24     }
25 }
```

Unlike DOM parser API, the static method `SchemaFactory.newInstance()` is used to instantiate the `SchemaFactory` object in SAX. The `Schema` instance is created by invoking `SchemaFactory.newSchema()` method. The `newSchema()` method can contain any validation schema (i.e. DTD, XML Schema). This schema object is used to define the structure of XML document and fetch appropriate data from XML data.

The static `SAXParserFactory.newInstance()` method is used to create `SAXParserFactory` object. This object has three options: `setvalidating()`, `setNamespaceAware()` and `setSchema()`. First two methods have a boolean argument. The `setValidating()` method with true argument indicates that the DOM parser must validate the XML document. The `setNamespaceAware()` with true argument specifies that the DOM parser will process the namespace names of validation mechanism (i.e. DTD or XML Schema) and the element tag names. Finally, the `setSchema()` method with a schema argument is invoked to `SAXParserFactory` to parse the XML document properly.

The `SAXParser` object is instantiated using `newSAXParser()` method which is invoked with `SAXParserFactory` object. The `SAXParser` object calls `parse()` method with two arguments. The first argument is the filename of XML document and another is the `SAXHandler` object. The `SAXHandler` class is implemented by the programmers. When an event occurs, the SAX Parser notifies the `SAXHandler` class to process the event. `startElement()`, `endElement()` and `endDocument()` methods are implemented into SAX Handler class to process their relevant events. When the parser gets an event of the starting tag of an element, the parser invokes the `startElement()` method of the handler to treat that event. If the parser has the ending tag of an element, the `endElement()` method is invoked will be called. Furthermore, the parser calls the `endDocument()` method, when the parser receives the ending tag of XML document.

The summary of the considered criteria for Simple API for XML as a desired XML parser is illustrated in Table 4.17.

XML document processing in SAX parser is difficult compare to DOM. SAX parser allows top-down serial access to the XML document, but it does not allow the random access to a particular element or content in XML document. SAX parser loads only a node as an event at a time into memory rather than the entire document. Thus it is a memory-efficient event-based parser. Since a node is accessed at a time and SAX does not maintain the information of previous nodes, it cannot create the tree structure into memory. However, SAX parser can only parse (unmarshalling) XML document but cannot create (marshalling) XML document. Unlike DOM parser, it can process both validated and non-validated document. The implemented SAX parser does not require considerable changes to reuse into other software applications. SAX parser is not allowed to change the existing schema components because of writing limitation in XML document. Object Relational Model (ORM) is very efficient and powerful mechanism to process and manipulate data as class objects from relational database and XML document. SAX parser does not afford this mechanism because it processes the nodes as events not as class objects.

XML Parser / Criteria	Simple API for XML (SAX)
Simplicity	Not simple
Flexibility in Data Access	Inflexible
Memory Efficiency	Efficient
Efficient Tree Structure	No
Marshalling and Unmarshalling	No
Validation Assurance	Yes
Portability	Yes
Customization	No
Object Relational Model Facility	No
Increased Productivity	No
Performance and Efficiency	Good
Flexibility for Non-XML Programmers	Inflexible
Validation Support on Demand	No

Table 4.17: Summary of criteria for SAX parser

In order to process and manipulate XML document in SAX, programmers require more lines of code which is time consuming like DOM (see Listing 4.7). SAX parser is a faster API because of less memory usage and traversing one node at a time. When XML document is being parsed, events are triggered in the SAX parser. In order to process and manage the events, a Handler class is implemented which is an overhead for the programmers. Furthermore, the programmers should also have sufficient knowledge on the structure and processing mechanisms of XML document for accessing and manipulating the documents in SAX parser (see Listing 4.7). SAX parser cannot validate the existing schema component on demand.

4.4.3 Streaming API for XML (StAX)

StAX stands for Streaming API for XML, which is another parser API for reading and writing XML documents. It is defined by JSR-173 of Java Community Process. It is an *event-driven pull-parser*. It has a programmatic cursor point which represents the cursor position in XML document. It treats XML document as a series of events. Unlike SAX parser, StAX parser forwards the cursor through XML document to fetch the parsed data. The main difference between SAX and StAX is that SAX parser pushes the parsed data to the software application, whereas StAX parser pulls the parsed data when the

application requires. Another important difference is SAX supports validation but StAX does not. StAX parser has two separate APIs. They are 1) Cursor API and 2) Event Iterator API.

Any of two APIs can be used by the software applications. Figure 4.8 represents the XML document processing mechanism in StAX.

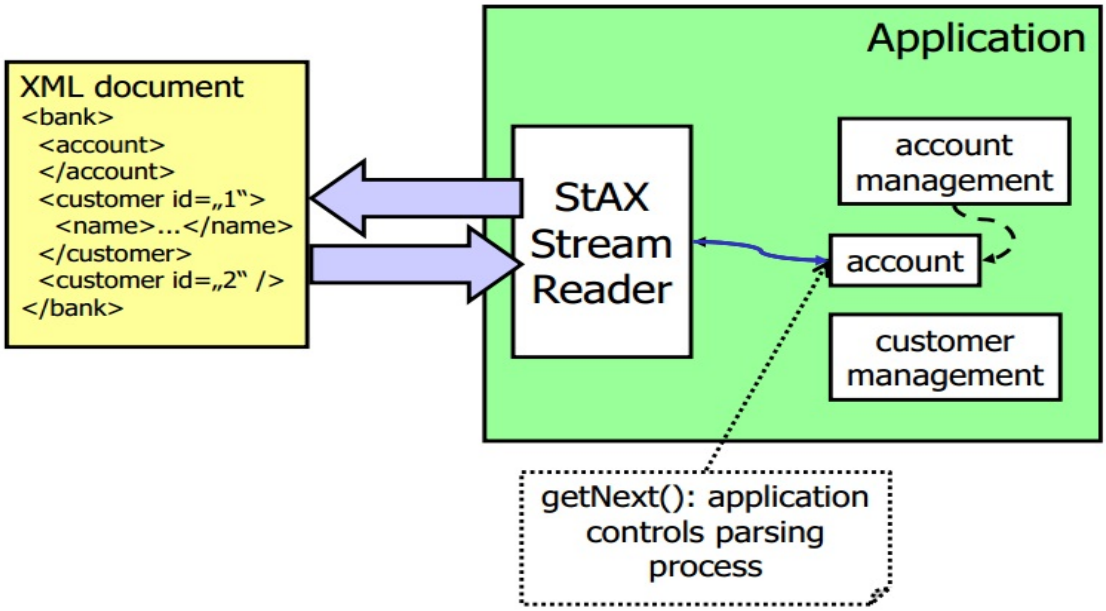

Figure 4.8: StAX API Architecture.

Source: Net-based applications lecture, Uni-Stuttgart

StAX parser also parses XML document "node by node", and every node represents an event like SAX. `START_ELEMENT`, `END_ELEMENT`, `CHARACTERS`, `ATTRIBUTE` etc. are the events of StAX parser. `XMLStreamReader` parses XML document and the cursor moves forward on the parsed XML document to process the retrieved event for the software application. However, the application pulls the event from the parser whenever it requires. On the other hand, `XMLStreamWriter` is used to generate the XML documents.

Listing 4.8 demonstrates how StAX parses and processes XML document.

Listing 4.8: Processing to parse data from XML document using StAX parser.

```
1 public class StAXParserExp {
2     public static void main(String[] args) {
3         XMLInputFactory factory = XMLInputFactory.newInstance();
4         factory.setProperty("javax.xml.stream.isNamespaceAware", false);
5         try {
6             XMLStreamReader reader = factory.createXMLStreamReader(new FileReader(new↩
                  File("agents.xml")));
```

```
7             while(reader.hasNext()) {
8             int event = reader.getEventType();
9             switch (event) {
10            case XMLStreamConstants.START_ELEMENT:
11                if(reader.getLocalName().equals("agentName"))
12                    System.out.println("The agent name: " + reader.getText());
13                break;
14            case XMLStreamConstants.END_DOCUMENT:
15                System.out.println("End of the document.");
16                break;
17            case XMLStreamConstants.ATTRIBUTE:
18                System.out.println("The attribute name: " + reader.getLocalName());
19                break;
20            }
21            reader.next();
22            }
23            reader.close();
24
25        } catch (Exception e) {
26            e.printStackTrace();
27        }
28    }
29 }
```

The name of Cursor API indicates that the StAX parser uses a cursor that moves forward from the beginning to end of XML document. The parser cannot store the previous states of the events and the cursor also cannot move backward. For this reason the parser is called *state independent*.

`XMLInputFactory` class is used to instantiate the XML stream reader instance. The `XMLInputFactory.newInstance()` static method is invoked to create `XMLInputFactory` instance. The `XMLStreamReader` interface parses XML document in a forward direction. In order to create `XMLStreamReader`, the `XMLInputFactory.createXMLStreamReader()` method is invoked. The `createXMLStreamReader()` method contains `FileReader` instance as an argument which has the XML document path as an argument. The `XMLStreamReader` instance processes a single event at a time. The first event of an XML document is `START_DOCUMENT` and the last one is `END_DOCUMENT`. The `XMLStreamReader.next()` method is used to forward to the next event in the parser. Since StAX does not support validation, this example does not include any validation interface.

The `getEventType()` method returns the type of an event. For example, `START_DOCUMENT`, `START_ELEMENT`, `ATTRIBUTE` etc. If event type is `START_ELEMENT` and the name of tag is

"agentName", the following output will be displayed.

```
The agent name:  WAB
```

The summary of the considered criteria for Streaming API for XML as an appropriate XML parser is illustrated in Table 4.18.

Criteria \ XML Parser	Streaming API for XML (StAX)
Simplicity	Not simple
Flexibility in Data Access	Inflexible
Memory Efficiency	Efficient
Efficient Tree Structure	No
Marshalling and Unmarshalling	Yes
Validation Assurance	No
Portability	Yes
Customization	No
Object Relational Model Facility	No
Increased Productivity	No
Performance and Efficiency	Good
Flexibility for Non-XML Programmers	Inflexible
Validation Support on Demand	No

Table 4.18: Summary of criteria for StAX parser

Processing of XML document in StAX parser API is relatively easier than SAX. Data access mechanism in StAX is almost same as SAX. This means StAX parser accesses data from XML document in top-down serial manner but cannot access data randomly. In order to use the memory efficiently, StAX resides only a node as an event at a time into memory instead of the whole document. StAX parser does not create tree structure of documents for processing and manipulating data. Moreover, software applications not only read XML document but also create XML document using StAX parser. Validation mechanism is not available in the StAX parser. The parsing program in StAX is portable. Unlike SAX, it does not require considerable changes to use to other software applications. Since StAX does not support validation, there is no way to change the existing schema and StAX does not provide ORM facility (from Lecture of Dr. Frank Duerr in Net-base Applications). In order to process and manipulate XML document in StAX, the programmers require more lines of code which is time consuming like DOM and SAX(see Listing 4.8). However, StAX parser is a fast API because of less mem-

ory usage and traversing one node at a time alike SAX. When XML document is being parsed, events are triggered in the StAX parser. There are several methods in StAX to process and manipulate the required data which are difficult to remember the usage of all methods by the programmers. Furthermore, the programmers should also have sufficient knowledge on the structure and processing mechanisms of XML document for accessing and manipulating data in StAX parser (see Listing 4.8).

4.4.4 Java Architecture for XML Binding (JAXB)

Data binding is an XML processing technique that works on Java classes instead of elements and attributes. That means, the schema either XML Schema or DTD defines the structure of an XML document and the data binding framework generates the Java classes regarding the schema. The data binding framework is used to build Java classes from a schema and vice versa. Several data binding frameworks are available in Java. Java Architecture for XML Binding (JAXB) will only be discussed specifically in this subsection.

Java Architecture for XML Binding (JAXB) is Java standard API which provides a fast, convenient and efficient data binding mechanism for converting XML document into a tree of Java objects and Java objects to XML document. JAXB has two powerful methods: *unmarshalling* and *marshalling*. The unmarshalling method reads the XML document into Java class objects and the marshalling method is used to write XML document from Java object tree.

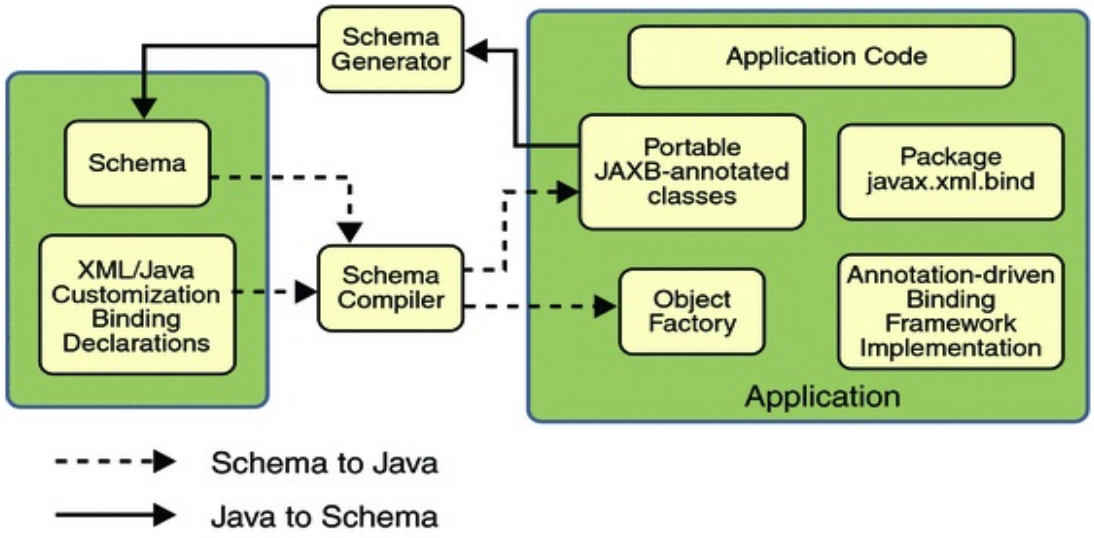

Figure 4.9: JAXB Architecture.

Source: http://docs.oracle.com/javaee/5/tutorial/doc/bnazg.html

Different components of JAXB are shown in Figure 4.9. These components define and build the JAXB implementation.

A JAXB implementation consists of the following architectural components:

Schema compiler: It binds the XML document into a tree of Java class objects.

Schema generator: It generates the XML schema from Java classes. In order to generate XML schema, Schema Generator uses the program annotations to make sure the correct order of the XML schema.

Binding runtime framework: It provides two important and powerful features: unmarshalling to read XML document and marshalling to write, manipulate and validate the XML schema from Java class representations.

Figure 4.10 describes the binding mechanism in JAXB.

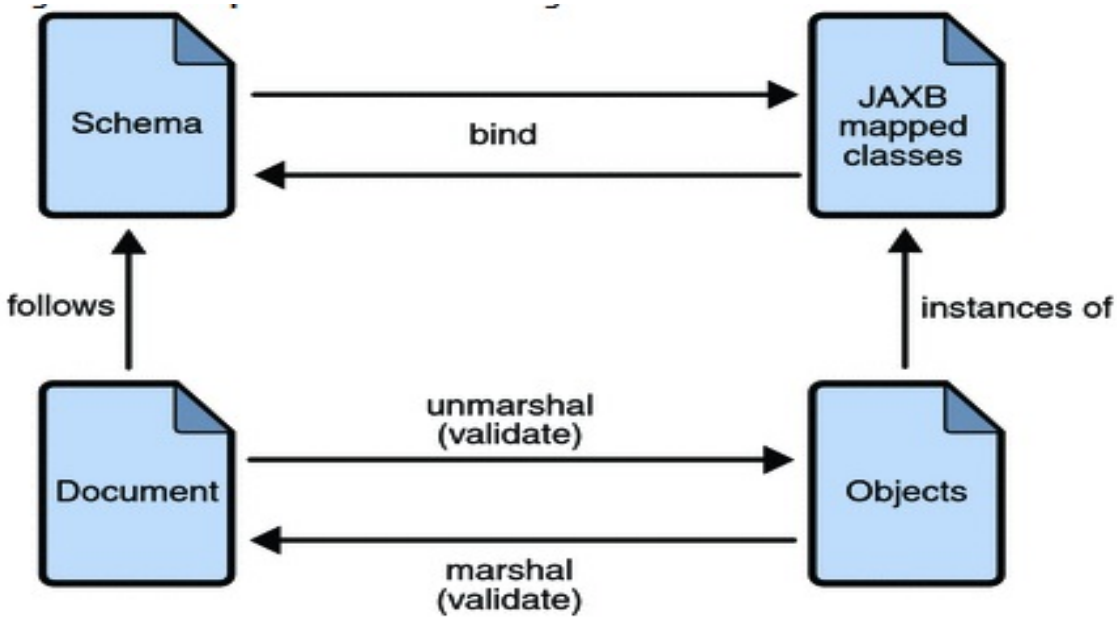

Figure 4.10: Steps in the JAXB Binding Process.

Source: http://docs.oracle.com/javaee/5/tutorial/doc/bnazg.html

JAXB maintains the following seven steps to process the data binding:

Generate classes: Schema Generator generates JAXB classes from XML schema. XML schema is used as input to create JAXB classes.

Compile classes: In order to execute the software application, the generated JAXB classes, source files and code of application must be compiled.

Unmarshal: XML document must follow the constraints which are defined in XML schema. The JAXB data binding framework uses unmarshall method to represent an XML document to Java class objects.

Generate content tree: The unmashall method generates a content tree of data objects which is delivered to the software application to process the input data.

Validate: The unmarshall mechanism validates the XML document when it is converted to a content tree. Validation is an optional operation to generate a content tree. If the content tree is modified, JAXB Validate operation can be used to validate the changes before creating XML document from Java objects.

Process content: The client application can modify the XML data represented by the Java content tree by means of interfaces generated by the binding compiler.

Marshal: This mechanism is used to generate XML document from Java objects. Validation operation may use in content before generating XML document.

JAXB generated files

JAXB compiler generates a class called `Agent` class from *agent.xsd* schema (Listing 4.9) into the target package.

Listing 4.9: XML Schema snippet to define an agent for generating POJO class.

```
<xsd:element name="agent">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="riskType" type="xsd:int"></xsd:element>
            <xsd:element name="name" type="xsd:string"></xsd:element>
    </xsd:complexType>
</xsd:element>
```

Basically the produced class is called a POJO (Plain Old Java Object) class. Listing 4.10 represents an example of generating class from XML Schema.

Listing 4.10: `Agent` POJO class example from *agent.xsd*.

```
@XmlRootElement(name = "agent")
public class Agent
{
    private int agentId;
    private String agentName;

    @XmlElement(name="agentId")
```

```
8       public String getAgentId() { return this.agentId; }
9       public void setId(int id) { this.agentId = agentId; }
10
11      @XmlElement(name="agentName")
12      public int getAgentName() { return this.agentName; }
13      public void setAgentName(String agentName) { this.agentName = agentName; }
14  }
```

`Agent` class consists of fields named *agentId* and *agentName* along with appropriate getter and setter methods of these fields. There are some metadata which are used for marshalling and unmarshalling of the class instance in JAXB.

Marshalling and Unmarshalling

When Java classes are generated from XML Schema, JABX parser can easily read XML document using unmarshalling mechanism and can write XML document from Java objects using marshalling mechanism. Listing 4.11 shows how to create an XML document from Java POJO classes and how to bind XML data into Java class objects.

Listing 4.11: Binding data from XML document to Java objects and writing data into XML document using JAXB.

```
1  public class JAXBParserExp
2  {
3      public static void main(String args[]) throws SAXException, JAXBException
4    {
5          try
6          {
7              Agent agent = new Agent();
8              agent.setAgentId(1);
9              agent.setAgentName("WAB");
10
11             JAXBContext context = JAXBContext.newInstance(Agent.class);
12             SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.↩
                   W3C_XML_SCHEMA_NS_URI);
13             Schema schema = factory.newSchema(new File("agent.xsd"));
14             // create XML document from Agent object.
15             Marshaller marshaller = context.createMarshaller();
16                       marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
17                    // read data from XML document.
18             Unmarshaller unmarshaller = context.createUnmarshaller();
19             unmarshaller.setSchema(schema);
20             Agent agent = (Agent) unmarshaller.unmarshal(new FileReader("agent.xml"))↩
                   ;
```

```
21              System.out.println("Agent Id: " + agent.getAgentId() + ", and agent name:↩
                    " + agent.getAgentName());
22          }
23          catch (Exception ex)
24          {
25              ex.printStackTrace();
26          }
27      }
28  }
```

The instance of `JAXBContext` is required to acquire the `Marshaller`, `Unmarshaller` and `Validator` interfaces. In order to create the `JAXBContext` instance, `JAXBContext` invokes with the static `newInstance()` method. In this example `JAXBContext.newInstance()` method contains `Agent` class as an argument, and the generation and conversion of XML document must follow the convention of `Agent` class.

Unlike DOM, SAX and StAX, the static method `SchemaFactory.newInstance()` is used to instantiate the `SchemaFactory` object in SAX. The `Schema` instance is created by invoking `SchemaFactory.newSchema()` method. The `newSchema()` method can contain any validation schema (i.e. DTD, XML Schema). This schema object is used to define the structure of XML document and fetch appropriate data from XML data. `Marshaller` class invokes `marshall()` method to create XML document from `Agent` class object. Running the program in Listing 4.11 generates the following output by the `Marshaller` class.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <agent>
3      <agentId>1</agentId>
4      <agentName>WAB</agentName>
5  </agent>
```

`Unmarshaller` class has overloading feature for `unmarshal()` method for different inputs. The `unmashall()` method is invoked into `Unmarshaller` class to fetch the data from *agent.xml* document. Running the program in Listing 4.11 produces the following output by the `Unmarshaller` class.

```
Agent Id:  1, and agent name:  WAB
```

The defined criteria are summarized in Table 4.19 for Java Architecture for XML Binding (JAXB) as a XML parser.

JAXB provides a tool to generate Java classes from a schema and vice versa. It can also generate XML document from Java content tree. Programmers do not need to

spend much time for creating, processing, manipulating XML document in XML-enable applications. Unlike DOM parser, JAXB creates Java content tree. For this reason, the software applications can access XML document sequentially and randomly. JAXB does not consider comments and processing instructions and in this way it protects the unnecessary memory usage. The memory efficiency is much better than DOM, but it is less memory-efficient than SAX and StAX.

XML Parser / Criteria	Java Architecture for XML Binding (JAXB)
Simplicity	Simple
Flexibility in Data Access	Flexible
Memory Efficiency	Efficient
Efficient Tree Structure	Yes
Marshalling and	Yes
Validation Assurance	Yes
Portability	Yes
Customization	Yes
Object Relational Model Facility	Yes
Increased Productivity	Yes
Performance and Efficiency	Good
Flexibility for Non-XML Programmers	Flexible
Validation Support on Demand	Yes

Table 4.19: Summary of criteria for JAXB.

Moreover, JAXB can easily bind XML data to Java class objects from XML document using `unmarshall()` method and can produce XML document from Java class objects by `marshall()` method. JAXB can be used for both validated and non-validated XML document. But it is recommended to use validated document in JAXB. JAXB ensures the validation in both creating and parsing XML document. Unlike other XML parsers, programmers do not need to make significant changes in the components of JAXB to use into other software applications. JAXB not only customizes XML document but also it can customize the XML schema and Java classes. Annotations in JAXB are used to customize the XML schema and Java classes. Object Relational Model (ORM) is very efficient and powerful mechanism to process and manipulate data as class objects from relational database. JAXB provides this feature with POJO classes along with appropriate getter and setter methods. Since data binding framework in JAXB generates Java classes from a schema automatically, it saves the programmers' time and the cost of the

software applications as well as the programmers do not require to have depth knowledge on XML language. The performance is significantly good for both small and large amount of data. JAXB can validate the content tree corresponding to an XML document, if it is necessary to validate that tree against the schema constraints on runtime.

4.4.5 Evaluation of Parsers

Earlier different parsers for XML document have been deliberated and investigated in detailed. Table 4.20 demonstrates the summary of all of the discussed parsers with defined criteria to choose an appropriate and suitable parser for processing the parameters of the AMIRIS model.

XML Parser / Criteria	DOM	SAX	StAX	JAXB
Simplicity	Simple for small file	Not simple	Not simple	Simple
Flexibility in Data Access	Flexible	Inflexible	Inflexible	Flexible
Memory Efficiency	Inefficient	Efficient	Efficient	Efficient
Efficient Tree Structure	Yes	No	No	Yes
Marshalling and Unmarshalling	Yes	No	Yes	Yes
Validation Assurance	No	Yes	No	Yes
Portability	Yes	Yes	Yes	Yes
Customization	No	No	No	Yes
Object Relational Model facility	No	No	No	Yes
Increased Productivity	Yes	No	No	Yes
Performance and Efficiency	Not good	Good	Good	Good
Flexibility for non-XML programmers	Inflexible	Inflexible	Inflexible	Flexible
Validation Support on Demand	No	No	No	Yes

Table 4.20: Summary of criteria for all parsers.

Selecting an appropriate parser with the fulfilment of all requirements for a specific software is really difficult. In order to select an appropriate and suitable Java parser, pro-

grammers should consider some important criteria according to the requirements of the software program. Table 4.20 shows that Document Object Model (DOM) is a useful and convenient parser to process and manipulate XML document. Although it meets most of the defined criteria to select as an appropriate parser for processing the parameters of the AMIRIS model, it does not fulfil some important criteria which are very crucial to optimize the code and to increase the performance of the AMIRIS model.

Table 4.20 demonstrates that Simple API for XML (SAX) is a memory efficient parser and it is useful to increase the performance of a software program. These two are not only criteria to select an appropriate parser for a software program. SAX is not able to maintain all of the defined criteria for processing the parameters of the AMIRIS model.

Like SAX, Streaming API for XML (StAX) is memory efficient and it increases the performance of a software. In addition, it is able to create XML document. But it does not fulfil most of the defined criteria to maintain the parameters of the AMIRIS.

Table 4.20 illustrates that Java Architecture for XML Binding (JAXB) is very easy to create Java classes from XML Schema and vice versa. Although JAXB is not memory efficient like SAX and StAX parser, it meets almost all of the defined criteria to select as an appropriate and suitable parser for processing the parameters of the AMIRIS model.

Finally, research and investigation evaluate that Java Architecture for XML Binding (JAXB) satisfies nearly all of the deliberated criteria. So JAXB has been selected to process the parameters for defining the model criteria and for initializing different agents of the AMIRIS model.

4.5 Batch Processing

Processing is the activity for organizing and managing of a particular task with required input data to produce output.

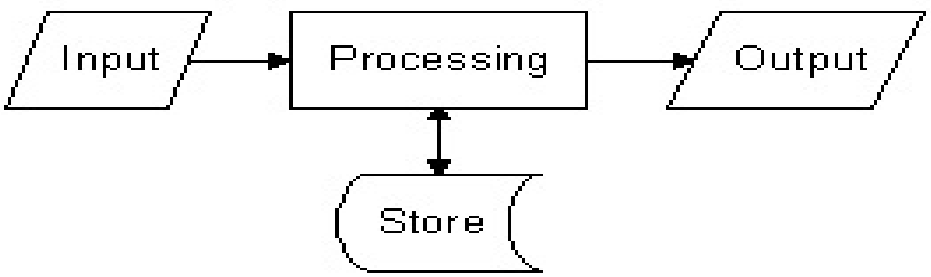

Figure 4.11: A Simple Processing.

Source: Author's illustration

Processing can be classified into three categories: *batch processing, transaction processing* and *interactive processing*. Figure 4.11 shows a basic processing mechanism. The software application takes the input. The input is processed with the business logic in the process unit to generate the expected output. In addition, the process unit stores and retrieves the further necessary data into/from storage device. This section describes batch processing.

Batch processing is a mechanism for defining, implementing and executing batch jobs with minimal user interaction. Most of the enterprise software applications are required to execute tasks without user communication. The tasks can be executed either periodically or when the applications use the small amount of resources. Billing, report generation and data format conversion are the examples of batch processing. The tasks are known as batch jobs. Most of the software tools provide a batch processing framework to execute batch jobs on a computer system. Batch processing gives the facility to the developers to concentrate only on the business logic to develop batch processing enabled applications. The batch processing framework comprises of batch job specification language, annotations and a set of interfaces that are used to implement business logic, manage the scheduling and execution of batch jobs as well as provide the interaction mechanism with batch container.

4.5.1 The Architecture of Batch Processing

The architecture of batch processing is a combination of processes and components which is used to manage the execution of batch jobs in correct and efficient way.

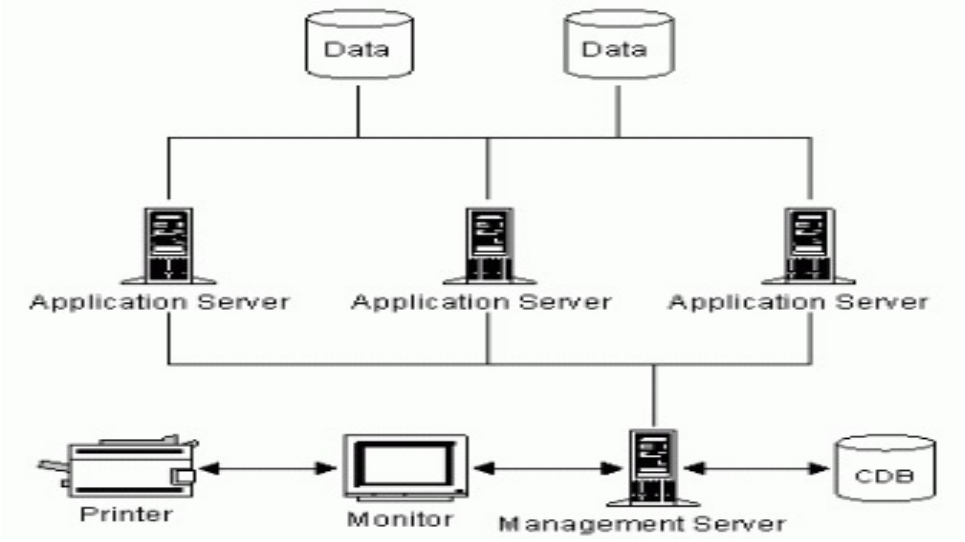

Figure 4.12: The Architecture of Batch Processing.

Source: http://www.microsoft.com/taiwan/technet/itsolutions/techguide/msm/smf/smfjobsc.mspx

Figure 4.12 shows an example of a batch architecture with different components. The main reason to design an efficient batch architecture is the optimization of a batch processing by performing the batch runs. A batch architecture should provide the facility for scheduling, monitoring, and controlling and error correction of the batch processing.

The *management server* is the main component of the batch architecture. It manages the scheduling of batch jobs and automatic execution of predefined scheduled batch jobs. It has several important functionalities. Some of them are starting and stopping jobs based on time, maintaining job queues, tracking job status, prioritizing the job etc.

The *Capacity Database* (CDB) consists of one or more databases which is used to store the performance related information centrally. For example, the batch and error logs are stored in the CDB repository.

Application server is another important component of the batch architecture. A scheduling tool is installed in every application server to monitor and log the batch jobs information. The applications stores the data into database device and retrieves data from database in case of necessity.

The *monitor* directly interact with management server. The management server can manually control the batch process using monitor. The *printer* generates the operational reports to evaluate the performance and so on.

4.5.2 Batch Processing Hierarchy

It is prerequisite to know the hierarchy of the batch processing and the components of the batch script for executing the scheduled batch jobs. Figure 4.13 represents an example of a batch processing hierarchy.

Each batch run contains several autonomous batch jobs for scheduled execution iteratively. Many organizations execute several batch processing depending on the requirements. Every batch job follows some specific steps for correct completion of that job. This formulated batch job skeleton is required for the consistent execution of every batch job. It specifies the standardized code and job-related information for each batch job. For example, the skeleton provides some important actions such as notification for successful and unsuccessful job execution, deletion or update of data.

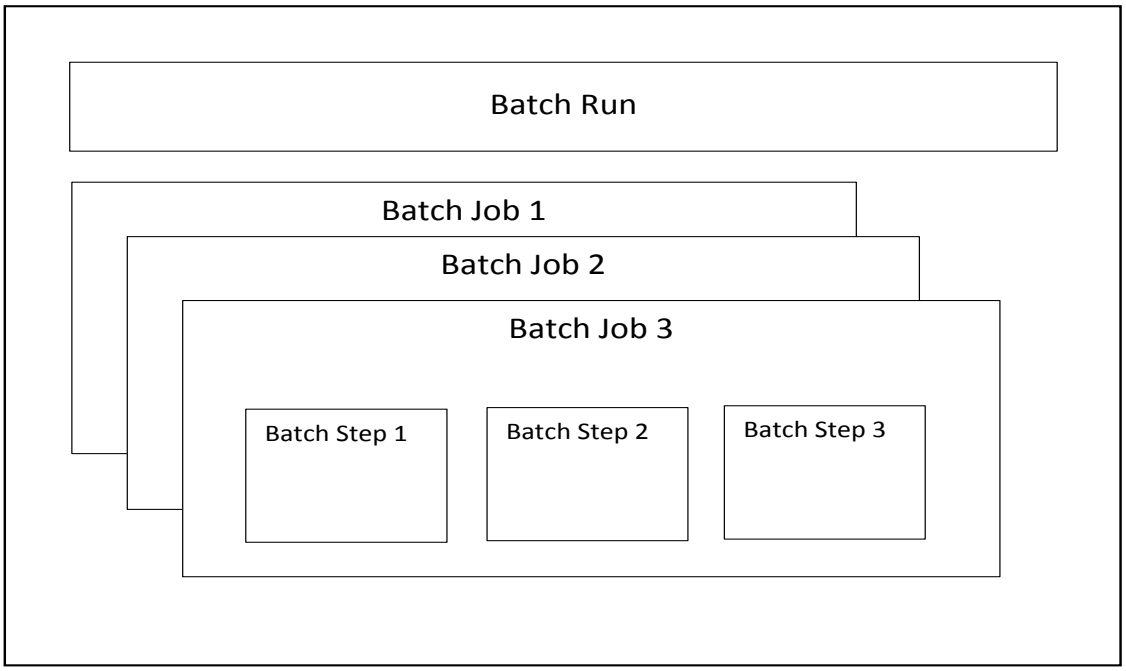

Figure 4.13: Hierarchy of Batch Processing.

Source: http://www.microsoft.com/taiwan/technet/itsolutions/techguide/msm/smf/smfjobsc.mspx

Figure 4.14 demonstrates the flowchart of a typical and scheduled batch process. The following flowchart gives the users a general high-level introduction and understanding of batch processing.

The scheduling tool schedules and initializes the batch jobs for performing the execution. It gives the priority to each batch job and orders the jobs corresponding to their priorities. If the execution of batch jobs does not start with scheduling priorities, the scheduling tool stops the execution and generates an error report. There are some schedulers which have the capability to restart the execution of batch jobs. After successful initialization of batch jobs, the first batch job is executed. The scheduler assigns one or more specific application servers (see Figure 4.12) to the batch jobs for performing the batch processing. After the error-free execution of the first batch job, the scheduler records a successful completion notification in the log file and the next batch job is executed, and so on.

The scheduler immediately terminates the execution and generates an error report, if it discovers any error during the execution of a batch job. A batch job can require different inputs for the successful execution. The particular input missing can be cause of the termination of a job execution. The scheduler also generates some warning messages during the job execution for different reasons. For example, if the memory overflow occurs for the batch job.

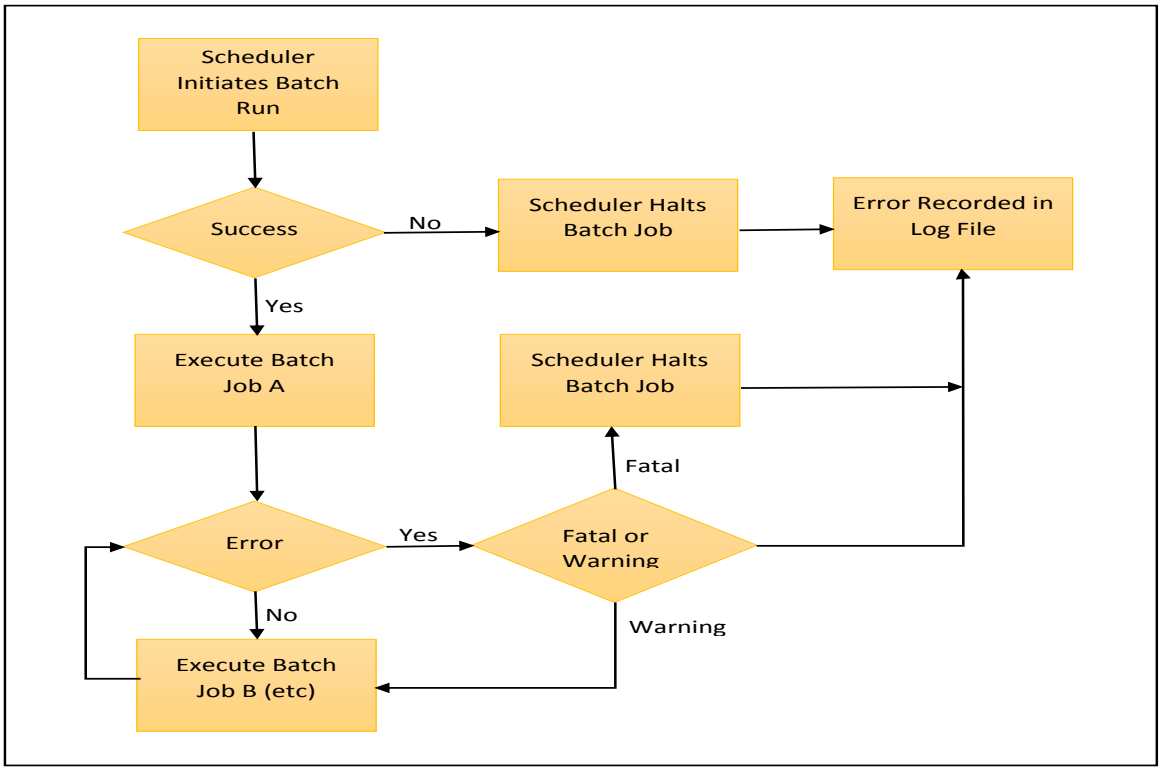

Figure 4.14: Flowchart of a typical Batch Processing.

Source: http://www.microsoft.com/taiwan/technet/itsolutions/techguide/msm/smf/smfjobsc.mspx

4.5.3 Batch Processing in Repast Simphony

Repast Simphony provides some powerful features to execute a model iteratively. The batch run in Repast Simphony is a set of individual executions of a model. The batch run defines own combination of parameters for each execution of the model. The user specifies the parameters for each execution and the number of executions of the model. Repast Simphony has the flexibility to perform these executions on local machine, on remote machine and in the cloud. Moreover, these executions could be accomplished on the combination of above three options or on a cluster. This section describes the batch run on local machine.

Repast Simphony 2.1 provides the facility to manage the batch run of a model using Batch Run Configuration GUI. But the AMIRIS model has been developed by Repast Simphony 2.0 which does not have this facility. So an external batch run application is required to perform the batch run of the AMIRIS model. After discussing external batch run application, batch run mechanism via Batch Run Configuration GUI with Repast Simphony 2.1 is described regarding a possible upgrade of the AMIRIS model to Repast

Simphony 2.1.

Batch Run using External Application

Since the AMIRIS model has been developed by Repast Simphony 2.0 with Java, it would be useful and convenient if the external application of the batch run can be implemented using Java. Repast Simphony provides the methods and interfaces to Java for implementing the external batch run application.

Figure 4.15 shows the flowchart how the external Java application processes the batch run in Repast Simphony.

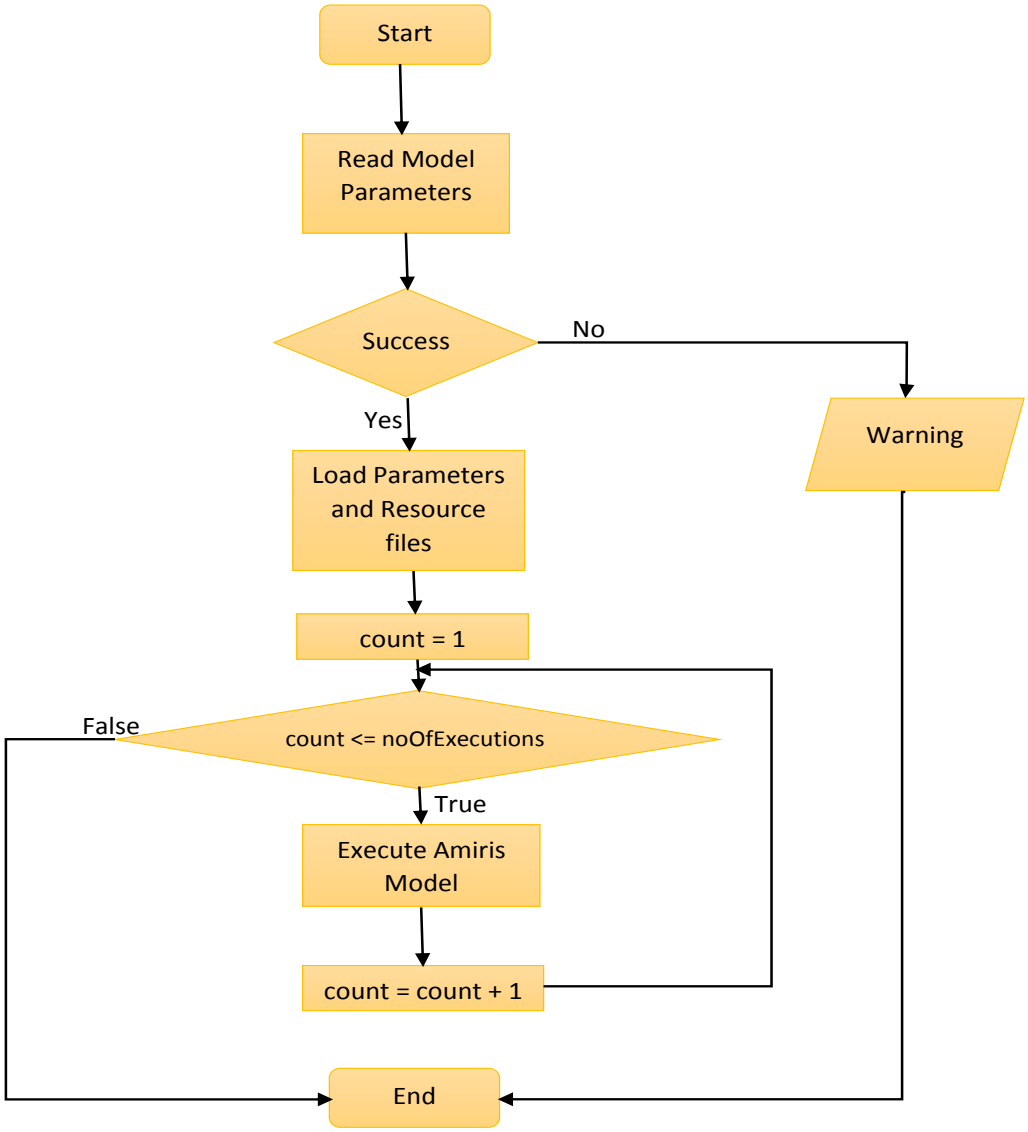

Figure 4.15: Flowchart Of External Batch Run Application.

Source: Author's illustration

The flowchart in Figure 4.15 shows that the external Java application begins with reading the parameters of the model criteria. If the application cannot read the parameters successfully, it stops the batch run and generates an error message. If parameter reading is successful, the loader loads the parameters of the agents and resource files to initialize the batch run. The user defines the number of executions in the external application for the model. After successful completion of all individual executions, the model is terminated.

The external Java application is robust for the batch run of the AMIRIS model. It requires less user interaction to manage multiple individual executions of the model. Additionally, manual configuration is not necessary like Batch Run Configuration GUI.

Batch Run using Configuration GUI in Repast Simphony 2.1

The batch run in Repast Simphony 2.1 can be run using Batch Run Configuration GUI. The Batch Run Configuration GUI configures and executes the batch run. The model is required to launch for having the configuration GUI.

Figure 4.16: Batch Run Configuration GUI Launch.

Source: Author's illustration

After clicking on the lightning button in Figure 4.16, the Batch Run Configuration GUI appears with seven buttons and four tabs as shown on the top of Figure 4.17.

The first five buttons are used for creating new batch configuration, opening existing batch configuration file, saving the batch configuration, saving the batch configuration into new file and updating the batch configuration. The last button is used to execute the batch run. Furthermore, in order to execute the batch run on the local machine, the options under *Model* and *Batch Parameters* tabs are required to configure and the options under the remaining tabs do not need to configure. The *Model* tab consists of *Model* and *Run* panels. The *Model* panel comprises of *Model Project, Scenario* and *Output Directory* properties. The *Model Project* property contains the location of the model. The *Scenario* property encloses the location of resource files of the model and

Output Directory holds the location to store the output of the model. Moreover, *Run* panel contains *SSH Key Directory, VM Arguments* and *Poll Frequency* properties. The *SSK Key Directory* encloses the public ssh rsa key of the user.[1] The virtual machine arguments are designed to give the instructions to the virtual machine to execute the model. The *Poll Frequency* defines the frequency with which the master process will poll its clients to determine if they have finished.

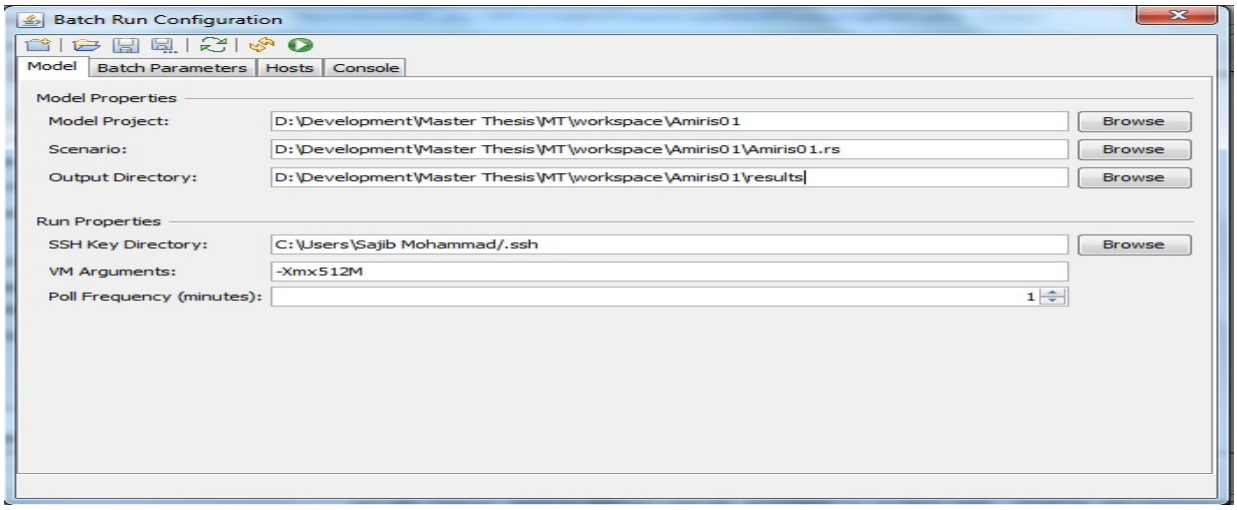

Figure 4.17: Batch Run Configuration GUI.

Source: Author's illustration

Figure 4.18 is displayed on the screen after clicking on *Batch Parameters* tab.

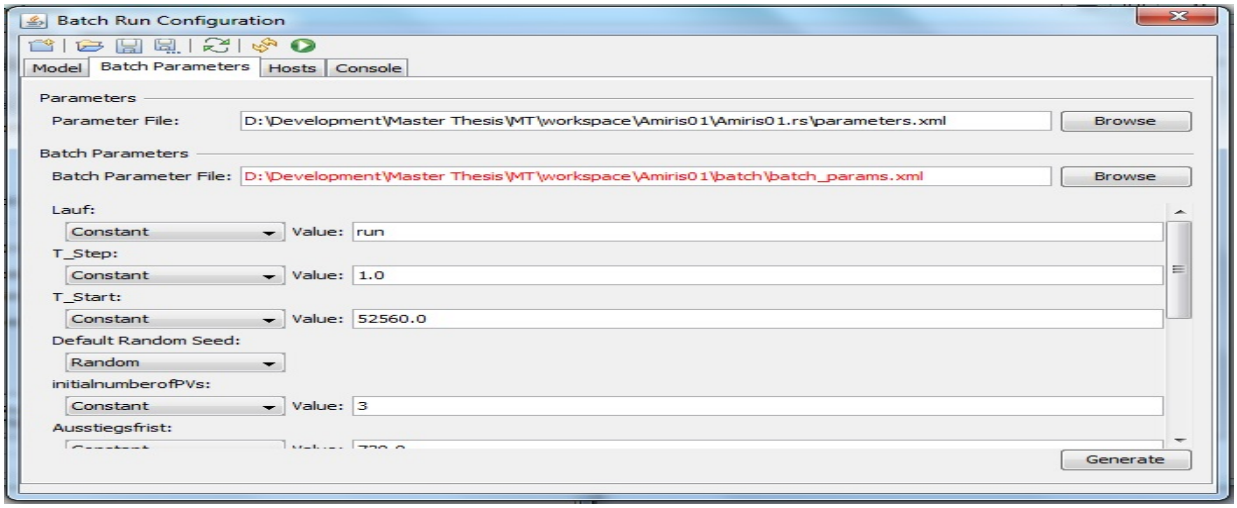

Figure 4.18: Batch Parameters Panel.

Source: Author's illustration

[1]SSH is a cryptographic network protocol for secured data communication between different remote computers.

Parameter File contains the non-batch parameter file location of the model. The parameters in the parameter file are used to define the model. This file is stored in the scenario directory of the model. The parameters of the batch run are stored in the *Batch Parameter File*. This file is in the batch directory.

Chapter 5

Prototyping and Implementaion

In software development, a prototype is a model of the earlier stage of a product or information system. It is developed to represent the purposes of the product or as a part of the development procedure. The user requirements are used to build a prototype. By the development of a prototype of a system, the user can better understand the system, and can add new requirements and modify the existing requirements to get the desired system. Normally, prototype is a useful process for developing the complicated and large systems (i.e. word processing software) which do not have pre-determined requirements. Before going to implement *parameterization* and *batch processing* for the AMIRIS model, a prototype is built to better realize the fulfilment of all requirements of the system. Section 5.1 gives a brief overview of the prototype for the AMIRIS model.

Software implementation or development is a process to develop a concrete software product from the specified requirements. The implementation methodology is not only used to write the code but also includes the preparation of requirements and objectives, the design of what is to be coded and confirmation that what is developed has met objectives. Section 5.2 demonstrates the implementation of *parameterization* and *batch processing* for the AMIRIS model and in section 5.3 results are shown to confirm the successful implementation.

5.1 Prototyping

Prototype is a blueprint of a software product which is used to test and evaluate a design of the software. This section presents the system architecture of the AMIRIS model and different components of the system as well as describes necessary components of batch processing for the AMIRIS model.

5.1.1 System Architecture

AMIRIS has been implemented for analyzing and investigating the direct marketing and market integration of renewable energy. The main important parts of this model are the parameterization of the model criteria and the agents as well as the setup of the model environment with different agents. To setup the parameterization of the model criteria, the user can use either the graphical user interface or XML file. In contrast, initializing the agents with their required parameters XML document is used only.

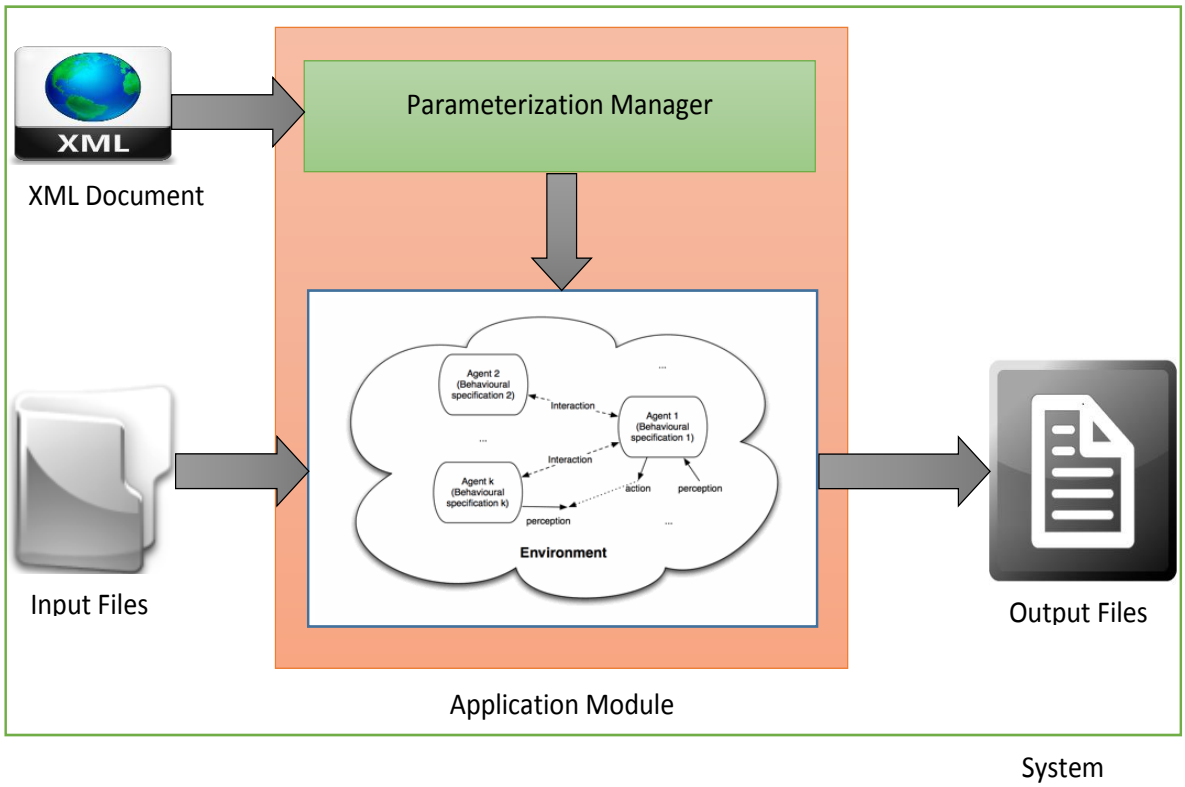

Figure 5.1: System Architecture of the AMIRIS model.

Source: Author's illustration

The system has the ability to produce the new parameters of the model criteria and the agents for the further usage. The newly generated parameters are stored into XML documents. The system defines an environment including different agents. Every agent has some properties to identify itself, communicate with other agents and take the actions. As a result the overall procedure delivers an agent-based modeling simulation system. Figure 5.1 represents the system architecture of the AMIRIS model.

Different software tools have been used to develop this system. They are included below:

– Repast Simphony 2.0

- Eclipse 3.6
- JDK 1.6
- JAXB 2.0
- Data Storage Application (Extensive Markup Language)

The following subsections describe different components of the system architecture.

5.1.1.1 Data Storage Application (Extensible Markup Language)

Data storage application is used to store the parameters of the model criteria and the agents. In order to access and store parameters efficiently, Extensive Markup Language (XML) has been used in this system. In AMIRIS model different XML documents are used to store different types of parameters. *parametersForBatchProcessing.xml* contains the parameters for defining the model criteria and *agents.xml* stores the parameters for instantiating the agents. These XML documents are stored in the *xml* folder of the project.

5.1.1.2 Application Module

The application module has two important parts: *parameterization of the model criteria and the agents*, and *environment setup*. The parameterization manager reads data from XML documents and data is being processed into the system for specifying the model criteria and for initializing different agents. On the other hand, in the environment module the system specifies an environment to enable communication among different agents and take the necessary actions by the agents. The user also provides necessary inputs to the agents for receiving expected output. The user defines the following properties to create an agent.

- ID of the Electricity Communication Object
- The final Electricity Buyer as partner agent
- The Agent class
- ID of Electricity Producer
- ID of the Intermediary
- ID of the Electricity Buyer
- The electricity share of physical flows to the contractors
- The electricity share is traded with the partner

- The shared amount of initial traded electricity
- The commercial path of Intermediaries
- The proportion of flexible generated capacity (for Biomass Plant Operator)
- Risk type parameter

5.1.1.3 Input and Output Files

Because of the detailed modeling capabilities in agent-based simulation models, extensive input data are needed. These data must be carefully evaluated in their potential development precisely in terms of their future developments. The AMIRIS model relies for most of the required data on the scenarios of the BMU Lead Study. Most of the input data are given only in time-discrete variables for the simulations in the AMIRIS model which are linearly interpolated or extrapolated. The same applies to their own empirical surveys on direct marketing to describe and analyze the potential future developments. All input and output data of the model using the real prices of each year. All input files are stored in the *data* folder of the project and output files are stored into *results* folder of the project.

5.1.2 External Java Batch Run Application

Repast Simphony 2.0 does not provide Batch Run Configuration GUI or any internal mechanism which can directly manage the multiple individual executions of the model in a single run (Chapter 4). But it includes necessary interfaces and features which can be used by the external application to implement the batch run of a model. An external Java Batch Run Application has been developed for the iterative execution of the AMIRIS model. To execute multiple individual executions of the model, external Java application defines the model criteria, initializes the agents, and creates the environment and adds the agents to the environment in each individual execution.

Adding and initializing the parameters of the model criteria, loading model scenarios with context, agent initialization via parameterization and adding agents to the context are the main parts of external Java application to manage the recursive execution of the AMIRIS model.

First of all, the batch run application reads the parameters which are used to define the model criteria. Then it loads the scenario file which contains different scenarios of the model and every scenario contains a context of the model which is used to add the agents into the model environment. Afterwards, the application initializes the agents via param-

eterization and add the agents to the environment using `AmirisContext` interface. The external application fetches *scenario.xml* and *context.xml* files from the folder *Amiris01.rs* of the AMIRIS project to load different scenarios with their context. The parameters for the model criteria and the agents as well as the input of the model (Section 5.1.1.3) can be same or different for each individual execution of the model.

5.2 Implementation

Different data storage applications, XML validation mechanisms and XML parsers have been analyzed and evaluated in Chapter 4. Furthermore, in Chapter 4 the batch processing is explored to execute the AMIRIS model repeatedly in a single run. The following sections describe the implementation of parameterization and batch processing of the AMIRIS model. The implementation is divided into two parts: *implementation of parameterization* and *implementation of batch processing*.

5.2.1 Implementation of Parameterization

This section introduces the necessary plug-ins and procedures for the development of the parameterization of the AMIRIS model. In this development part, the setup of different criteria of the model and initialization of agents are implemented.

JAXB 2.0 plug-in is used for directly generating Java POJO (Plain Old Java Object) classes from XML Schema and XML Schema from Java classes (Section 4.4.4 in Chapter 4). It can be included into Eclipse IDE (Integrated Development Environment) easily. Figure 5.2 shows the JAXB plug-in with its included wizards.

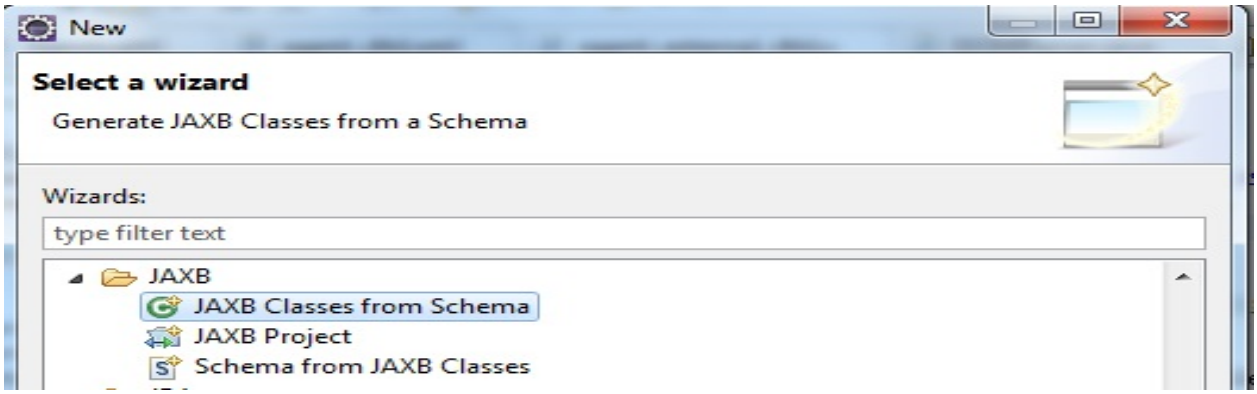

Figure 5.2: JAXB Plug-in.

Source: Author's illustration

JAXB Classes from Schema: This wizard is used to generate Java classes from XML Schemas automatically.

JAXB Project: It helps to create Java classes from XML Schema and vice versa.

Schema from JAXB Classes: This wizard creates XML Schema from Java classes.

Section 5.2.1.1 illustrates the creation of XML Schemas for the parameters of the model criteria and of the agents. Section 5.2.1.2 describes the Java classes generation from XML Schemas. Parser implementation is presented in Section 5.2.1.3. Section 5.2.1.4 shows how to add the parameters to the AMIRIS model for defining the model criteria and section 5.2.1.5 describes how an agent is initialized. Finally section 5.2.1.6 demonstrates the process for adding the created agents to the AMIRIS Context.

5.2.1.1 XML Schema Creation

Although JAXB plug-in can generate both Java classes from XML Schema and XML Schema from Java classes, initially the developers must create either XML Schema or Java classes for generating one from another. In that case, the developers should analyze and investigate which is cost and time effective to create at first. Since XML Schema requires few line of coding and is time-effective, it is created first to produce Java classes for this implementation. The XML Schemas are used to validate the parameters and specify the structure of XML document as well as generate the Java classes for defining the criteria of the model and initializing the agents in the model. Listing 5.1 shows the snippet of *parametersForBatchProcessing.xsd* which is used to define and validate the parameters of the model criteria.

Listing 5.1: Defining and validating the parameters of the AMIRIS model to setup the model environment.

```
<xsd:element name="parameters">
<xsd:complexType>
    <xsd:sequence>
        <xsd:element name="param" minOccurs="1" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                <xsd:element name="name" type="xsd:string"/>
                <xsd:element name="classType" type="xsd:string"/>
                <xsd:element name="value" type="xsd:string"/>
                <xsd:element name="readability" type="xsd:boolean"/>
                </xsd:sequence>
            </xsd:complexType>
```

```
13          </xsd:element>
        </xsd:sequence>
15  </xsd:complexType>
    </xsd:element>
```

The element-name `parameters` is the *root* element of XML document and is defined as complex type (line 2 in Listing 5.1). The `param` element is also a complex type element and contains some simple type elements. Every simple type element has a data type. The `minOccurs` and `maxOccurs` facets define the occurrence boundary of a particular element. The `param` element must occur once and can appear several times in XML document. The `minOccurs` property sets value '1' which means the `param` element occurs at least once and the `maxOccurs` property sets the value 'unbounded' means that the `param` element can repeat unlimited times.

Listing 5.2 shows the snippet of *agents.xsd* which is used to define and validate the parameters of the agents.

Listing 5.2: Defining and validating the parameters of the agents for initializing the agents.

```
    <xsd:element name="agents">
2   <xsd:complexType>
    <xsd:sequence>
4   <xsd:element name="agent" minOccurs="1" maxOccurs="unbounded">
    <xsd:complexType>
6       <xsd:sequence>
            <xsd:element name="agentId" type="xsd:int" minOccurs="1" maxOccurs="1" />
8           <xsd:element name="agentName" type="xsd:string" minOccurs="1" maxOccurs="1"↩
                />
            <xsd:element name="kommStromList" minOccurs="0" maxOccurs="1">
10              <xsd:complexType>
                    <xsd:sequence>
12                      <xsd:element name="kommStrom" minOccurs="0" maxOccurs="unbounded">
                        <xsd:complexType>
14                          <xsd:sequence>
                            <xsd:element name="kommId" type="xsd:string" />
16                          <xsd:element name="agentenklasse" type="xsd:string" />
                            <xsd:element name="idSE" type="xsd:int" />
18                          <xsd:element name="idZWH" type="xsd:int" />
                            <xsd:element name="idSK" type="xsd:int" />
20                          </xsd:sequence>
                        </xsd:complexType>
22                      </xsd:element>
                    </xsd:sequence>
```

```
24              </xsd:complexType>
            </xsd:element>
26      </xsd:sequence>
    </xsd:complexType>
28  </xsd:element>
    </xsd:sequence>
30  </xsd:complexType>
    </xsd:element>
```

Likewise, the `agents` element is the *root* element of the XML document and is defined as complex type (line 2 in Listing 5.2). It contains a list of agents. The `agent` element is also a complex type element, and contains simple type and complex type elements. When an `agent` element occurs in the XML document, the `agentId` and `agentName` elements must occur only once because they have `minOccurs` and `maxOccurs` boundary restrictions which is set to value '1'. The `kommStromList` element is not mandatory to occur with `agent` element since `minOccurs` and `maxOccurs` properties set to '0' and 'unbound' respectively. The elements which do not have any boundary restriction, they must occur only once in the document.

5.2.1.2 POJO Classes Creation

After creating XML Schemas, Java classes are generated from the corresponding schema using JAXB plug-in. A Java class is produced for each and every complex type element of the schema. On the other hand, an instance class variable for each simple type element is generated along with appropriate getter and setter methods. The class variables will be declared automatically with the corresponding data types which are defined in XML Schema. Moreover, if an element contains the complex element, JAXB plug-in declares an object typed or list of an object typed class variable with getter and setter methods. The snippet of generating Java classes from *parametersForBatchProcessing.xsd* XML Schema is shown in Listing 5.3.

Listing 5.3: An example of generating Java classes to bind the parameters of the model criteria to the implemented JAXB parser

```
1 @XmlRootElement(name = "parameters")
2 public class Params
3 {
4     @XmlElement(name="param")
5     private List<Param> paramList = new ArrayList<Param>();
6
7     public List<Param> getParamList() { return this.paramList;}
```

```
8     public void setParamList(ArrayList<Param> paramList) { this.paramList = ↩
          paramList;}
9 }
```

`Param` class contains a set of fields with appropriate getter and setter methods which are used to bind the values of the corresponding parameters of the model criteria from *parametersForBatchProcessing.xml* document. On the other hand, `Parameters` class holds a list of parameters to define the model criteria.

Listing 5.4 represents the snippet of generating Java classes from *agents.xsd* XML Schema.

Listing 5.4: An example of generating Java classes to bind the agents' parameters to the implemented JAXB parser

```
1  @XmlAccessorType(XmlAccessType.FIELD)
2  @XmlType(name = "", propOrder = {
3      "agent"
4  })
5  @XmlRootElement(name = "agents")
6  public class AgentListDAO {
7
8      @XmlElement(required = true)
9      protected List<AgentDAO> agent;
10
11     public List<AgentDAO> getAgentList() {
12         if (agent == null) {
13             agent = new ArrayList<AgentDAO>();
14         }
15         return this.agent;
16     }
17 }
```

`AgentDAO` class includes `agentId`, `agentName` and `KommStromList` (incorporates a list of *electricity coommunication objects* (KommStrom), Listing 5.2) fields as well as other necessary properties with getter and setter methods. These methods are used to bind the values of the corresponding agent's parameters from *agents.xml* file. Finally `AgentListDAO` comprises a list of agents to add them to the AMIRIS Context.

5.2.1.3 Parser Implementaion

The JAXB parser fetches data from XML document. To retrieve parameters from *parametersForBatchProcessing.xml* and *agents.xml*, JAXB parser is implemented in the

AMIRIS model. `JAXBProcessor` class (in *amirishlp* package of the project) implements `readModelParametersFromXMLFile()` and `readAgentsWithParametersFromXMLFile()` static methods to fetch the parameters of the model criteria and the parameters of the agents respectively. Listing 5.5 describes the JAXB implementation of the parameterization for defining the model criteria and Listing 5.6 demonstrates the JAXB implementation of the agents' parameterization.

Listing 5.5: Retrieving the parameters for defining the model criteria from *parametersForBatchProcessing.xml* and returning a list of parameters to the external Java application.

```
1  public static List<Param> readModelParametersFromXMLFile() throws SAXException, ↩
       JAXBException
2  {
3      // Create JAXB Context.
4      JAXBContext context = JAXBContext.newInstance(Params.class);
5      // Create schema factory to validate the XML schema.
6      SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.↩
           W3C_XML_SCHEMA_NS_URI);
7      // Specify the location and name of the XSD file to validate the XML file.
8      Source schemaFile = new StreamSource(new File(Constants.FileNames.↩
           PARAMETERS_XSD_FILE));
9      // Create the schema to validate XML file.
10     Schema schema = factory.newSchema(schemaFile);
11     // Initialize Unmarshaller to create Java object from XML file
12     Unmarshaller unmarshaller = context.createUnmarshaller();
13     // Set the schema to validate XML file.
14     unmarshaller.setSchema(schema);
15     // Generate list of Java objects from XML file with defined values in ↩
           parametersForBatchProcessing.xml.
16     return ((Params) unmarshaller.unmarshal(new File(Constants.FileNames.↩
           PARAMETERS_XML_FILE))).getParamList();
17 }
```

In line 4, `JAXBContext` object is created and contains `Params` class. The `Params` should be the root element of *parametersForBatchProcessing.xml*. But it is possible to change the classname or fieldname in Java POJO classes using JAXB annotation for any element which is mentioned in Listing 5.3. JAXB creates the `Unmarshaller` object using the static `createUnmarshaller()` method of `JAXBContext` in line 12. The `Unmarshaller` interface creates a tree of Java objects. Finally the `unmarshal()` method is used to set the values from *parametersForBatchProcessing.xml* to the corresponding fields and the parameter list is returned to the external application for defining the model criteria in line 16.

Listing 5.6: Retrieving the agents with their corresponding parameters from *agents.xml* and returning a list of agents to the AMIRIS model.

```
public static AgentListDAO readAgentsWithParametersFromXMLFile() throws SAXException, JAXBException
{
    // Create JAXB Context.
    JAXBContext context = JAXBContext.newInstance(AgentListDAO.class);
    // Create schema factory to validate the XML schema.
    SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    // Specify the location and name of the XSD file to validate the XML file.
    Schema schema = factory.newSchema(new File(Constants.FileNames.AGENTS_XSD_FILE));
    // Initialize Unmarshaller to create Java object from XML file.
    Unmarshaller unmarshaller = context.createUnmarshaller();
    // Set the schema to validate XML file.
    unmarshaller.setSchema(schema);
    // Generate list of Java objects from XML file with defined values in agents.xml. */
    return ((AgentListDAO) unmarshaller.unmarshal(new FileReader(Constants.FileNames.AGENTS_XML_FILE)));
}
```

JAXB also creates a `JAXBContext` object with `AgentListDAO` class as an argument in Listing 5.6. The `unmarshal()` method uses *agents.xml* file as argument and, finally a list of agents is returned to the AMIRIS model for initializing the agents.

5.2.1.4 Adding and Initializing the Parameters of the Model Criteria

The previous sections have prepared the reading of the parameters for defining the criteria of the model and for initializing the agents. The procedure to add and initialize the parameters for defining different criteria of the model is only illustrated in this section with a code snippet.

Listing 5.7: Adding the parameters of the model criteria into AMIRIS model.

```
for(Param param : JAXBProcessor.readModelParametersFromXMLFile())
{
    // Setup all of the required parameters to initiate the model.
    Class<?> classType = Class.forName(param.getClassType());
    if(classType.equals(String.class))
        paramCreator.addParameter(param.getName(), classType, param.getValue(), param.isReadability());
```

```
7       else if(classType.equals(Integer.class))
8           paramCreator.addParameter(param.getName(), classType, Integer.parseInt(param↩
                .getValue()), param.isReadability());
9   }
10  // create a params object contains the added parameters.
11  params = paramCreator.createParameters();
```

In the code snippet of Listing 5.7, `JAXBProcessor.readModelParametersFromXMLFile()` returns a list of parameters and *for* loop is used to fetch each parameter from the list. The parameters can be any primitive data type such as integer, double, string, boolean. For this reason, in line 4 the *wildcard type* (i.e. <?>) class object has been used so that it can be compatible for any kind of Class object associated with the class or interface with the given string name. The code from line 5 to 8 is used to add all retrieved parameters to the `repast.simphony.parameter.ParametersCreator` class. In line 11 `createParameters()` method of `ParametersCreator` class creates an object of parameters which contains the added parameters.

Listing 5.8: Initializing the parameters to define the criteria of the AMIRIS model.

```
1   Controller controller = new DefaultController(runEnvironmentBuilder);
2   controller.runParameterSetters(params);
```

`BatchProcessingRunner` class (in *batchProcessing* package) instantiates the `repast.simphony.engine.controller.Controller` interface to incorporate the parameters and the scenarios of the model and to control the overall model. Moreover, the `Controller` interface uses `runParameterSetters(params)` method to initialize the parameters for the current execution of the model.

5.2.1.5 Agent Initialization

This section explores the initialization procedures of different agents for the AMIRIS model. The class `AgentInitialization` is used to instantiate the agents. The initialization of an agent is demonstrated with the Java code snippet in Listing 5.9.

Listing 5.9: Initializing the agents to add them into the AMIRIS Context.

```
1   for(AgentDAO agent : this.agents.getAgentList())
2   {
3       this.kommStromList.clear(); // clear old KommStromList
4       if(agent.getKommStromList() != null && agent.getKommStromList().getKommStrom().↩
            size() > 0)
5       {
```

```
6           KommStrom kommStrom = null;
7           // retrieve each KommStrom of relevant agent.
8           for(KommStromDAO kommStromDAO : agent.getKommStromList().getKommStrom())
9           {
10              // call KommStrom constructor with required parameters.
11              kommStrom = new KommStrom(kommStromDAO.getKommId(), null, kommStromDAO.↩
                    getAgentenklasse() + "", kommStromDAO.getIdSE(), kommStromDAO.↩
                    getIdZWH(), kommStromDAO.getIdSK(), kommStromDAO.getAnteilStromPhys()↩
                    , kommStromDAO.getAnteilStrommengeVermarktung(), kommStromDAO.↩
                    isActive());
12              this.kommStromList.add(kommStrom);
13          }
14      }
15      if(agent.getAgentName().equals("WAB")) // check the currently fetched agent is ↩
            WAB or not.
16      {
17          Wab wab = new Wab(agent.getAgentId()); // call WAB constructor with required↩
                parameters.
18          if(this.kommStromList.size() > 0) // check the length of KommStromList
19              for(KommStrom kommStrom : this.kommStromList) // retrieve each KommStrom ↩
                    object to add to WAB.
20              {
21                  wab.getKommList().add(kommStrom); // add KommStrom object to WAB.
22              }
23          this.agentList.add(wab); // add WAB object to agentList.
24      }
25  }
```

The snippet shows that `this.agents.getAgentList()` method returns a list of agents and *for* loop in line 1 retrieves an agent with its necessary parameters in each iteration. The communication objects of an agent are initialized in line 11. They are added into `KommStromList` object which is shown in line 12. Then agent is being checked whether it is *WAB* type agent. From line 17 - 22 an object of *WAB* agent is created and the communication objects are integrated with *WAB* agent. Finally *WAB* agent is appended to `agentList` object in line 23. The `agentList` will be used when the AMIRIS model have to add the agents in its context (environment).

5.2.1.6 Adding Agents to AMIRIS Context

The class `AmirisContext` contains the agents (i.e. wind, photovoltaic power plant operator, intermediary). The agents maintain several contacts and communications between

them to perform their desired actions in the AMIRIS model. After initialization of agents, it is very easy to add the agents to the AMIRIS Context. Listing 5.10 shows the procedure how the agents are added to the AMIRIS Context.

Listing 5.10: Adding Agents to AMIRIS Context

```
try
{
    // instantiate an AgentInitialization object.
    AgentInitialization agentInit = new AgentInitialization();
    // create an AmirisAgentsXMLDb object.
    AmirisAgentsXMLDb db = new AmirisAgentsXMLDb();
    //retrieve a list of agents.
    if(agentInit.getAgentList().size() > 0)
    {
        // read an agent in every iteration.
        for(SAgent agent : agentInit.getAgentList())
        {
            // add every agent to the AMIRIS Context.
            context.add(agent);
            // add every agent to AmirisAgentsXMLDb class to generate agents with ↩
                modified values of the parameters.
            db.getAgentList().add(agent);
        }
    }
}
// write all agents with their modified parameters into an XML document.
db.writeAgentList(dbFile);
```

The class `AgentInitialization` initializes all agents (Listing 5.9). The *for* loop is used to fetch every agent from the agent list. Every agent is added to `AmirisContext` object. `AmirisAgentsXMLDb` class (in *ext* package) implements `db.getAgentList()` method that stores all agents with their modified parameters. The `db.writeAgentList(dbFile)` method of `AmirisAgentsXMLDb` writes the agents into an XML document for further usage.

5.2.2 Batch Run Implementation

Repast Simphony 2.0 does not provide Batch Run Configuration GUI or any internal mechanism which can directly manage the multiple individual executions of the model in a single run (Section 5.1.2). For these reasons, an external Java batch run application has been implemented for the iterative execution of the AMIRIS model in a single run.

The batch run application reads the parameters, and adds and initializes them to the AMIRIS model. Reading of the parameters is described in Listing 5.5. Section 5.2.1.4 has demonstrated the adding and initializing procedure of the parameters. This section illustrates the implementation of the batch run.

5.2.2.1 Loading Scenario File

The *Amiris01.rs* folder of the project contains *context.xml* file to setup the model environment and *scenario.xml* file to load different scenarios of the model. In section 5.1.2 the scenario file has been described in detailed. The code snippet in Listing 5.11 describes how the scenario file is loaded into the model.

Listing 5.11: Loading Scenario File

```
1 RunEnvironmentBuilder runEnvironmentBuilder = new DefaultRunEnvironmentBuilder(this,↩
      true);
2 Controller controller = new DefaultController(runEnvironmentBuilder);
3 controller.setScheduleRunner(this);
4 BatchScenarioLoader loader = new BatchScenarioLoader(scenarioDir);
5 ControllerRegistry registry = loader.load(runEnvironmentBuilder);
6 controller.setControllerRegistry(registry);
```

The `repast.simphony.engine.environment.RunEnvironmentBuilder` class builds a run environment for the model. The user can customize the run environment for every individual execution of the model in case of necessity. The `repast.simphony.engine.controller.Controller` interface creates a controller for the simulation. This interface maintains the setup, execution and teardown of the model.

The `repast.simphony.batch.BatchScenarioLoader` class loads the scenario file so that the model can run in batch mode. Moreover, it loads the parameters to define the model criteria.

The `repast.simphony.engine.environment.ControllerRegistry` class is a registry that contains a set of controller actions. Finally the `setControllerRegistry()` method set the registry of the controller actions which the controller will run.

5.2.2.2 Iterative Execution of AMIRIS Model

The batch run application prepares the iterative execution of the AMIRIS model when the parameters are initialized and the scenario file is loaded successfully. This section

demonstrates how the external batch run application executes the AMIRIS model iteratively in a single run, and how the model is initialized in each execution.

Listing 5.12: Code Snippet of Iterative Execution of AMIRIS Model

```
// Run the Simulation model for 4 times.
for(int i = 0; i < ParamConsts.Running.NUMBER_OF_EXECUTIONS; i++)
{
    int tickCount = -1;
    runner.runInitialize(); // initialize the run of the model
    while (runner.getActionCount() > 0) // loop until last action
    {
        if (tickCount == (int)(endTime - startTime))
        {
            runner.setFinishing(true); // finish the execution of model.
        }
        tickCount++;
        runner.step(); // execute all scheduled actions at next tick
    }
    runner.stop(); // execute any actions scheduled at run end
    runner.cleanUpRun(); // run after one run complete
}
runner.cleanUpBatch(); // run after all runs complete
```

Listing 5.12 shows that an individual execution is performed of the model for each iteration of the *for* loop. The `BatchProcessingRunner` class calls the `runInitialize()` method to initialize the model before new execution. The `setFinishing()` method is used to end the execution of the model. For each tick, the `step()` method is invoked to the `BatchProcessingRunner` class for executing all scheduled actions in the model. The `stop()` method executes all the actions which are scheduled to accomplish at the end of the model execution. The `cleanUpRun()` method cleans up all actions after an execution of the model which is just occurred. Finally `cleanUpBatch()` method cleans all the actions after a batch run (a set of executions of the model in a single run).

5.3 Results

In the past the AMIRIS model was not able to manage iterative execution with different set of parameters in a single run. For every set of parameters the AMIRIS model was required an individual execution for producing the output. This means for n sets of parameters the AMIRIS was needed n individual executions for generating n different sets of output for better statistical analysis. For this reason, considerable amount of

time and more human interactions were required to operate the AMIRIS model which decreased the performance of the model and made the model cost ineffective. Figure 5.3 shows an abstract view of simulation process of the previous AMIRIS model.

The AMIRIS model was only able to take one set of parameters in a run and always produced one set of output. It was not able to perform with different set of parameters for producing different set of output in a single run. This means it was unable to perform iterative execution. For generating different set of output, it was required to change the parameters manually in the source before each run.

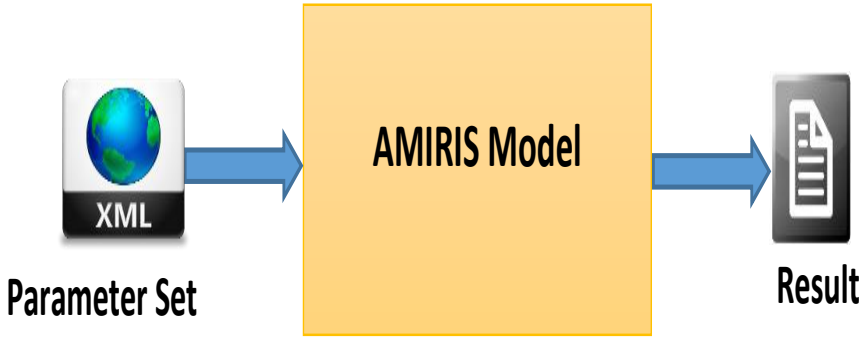

Figure 5.3: Abstract view of simulation process of previous AMIRIS model.

Source: Author's illustration

After developing dynamic parameterization and batch processing, currently the AMIRIS model is able to operate with different set of parameters in a single run. Now it can manage the iterative execution and in each iteration it can automatically adopt new set of parameters for generating new output. The AMIRIS model is also capable to produce different set of output with a single set of parameters in a single run. Figure 5.4 represents an abstract view for generating different set of output with different set of parameters in a single run.

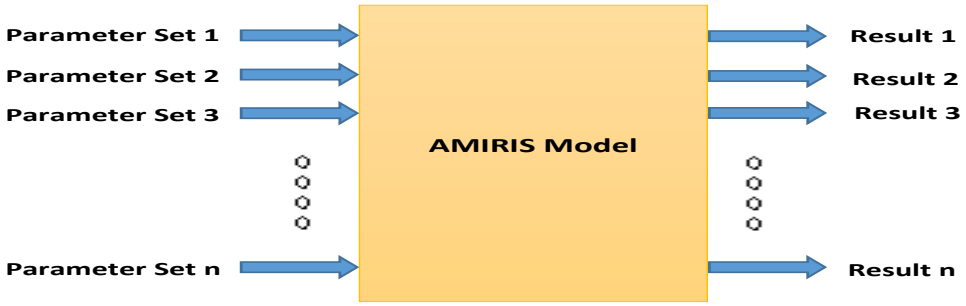

Figure 5.4: An abstract view of the AMIRIS model for generating different output with different set of parameters in a single run.

Source: Author's illustration

Since the AMIRIS model can manage iterative execution, it takes a set of parameters and produces a set of output in each iteration.

There are different power plant operators for generating electricity. Wind power plant operator has been taken as an example to analyze the annual income of the intermediary. Let's assume, the intermediary sells certain amount of total electricity of wind power plant operator to the market. The income demands on the share of electricity to the market. If the amount of electricity is increased, the income should be increased proportionally. Although the intermediary shares 0% of electricity of wind power plant operator to the market, the intermediary is able to earn from other power plant operators (i.e. photovoltaic, biomas). Figure 5.5 demonstrates the annual income of the intermediary for 2013.

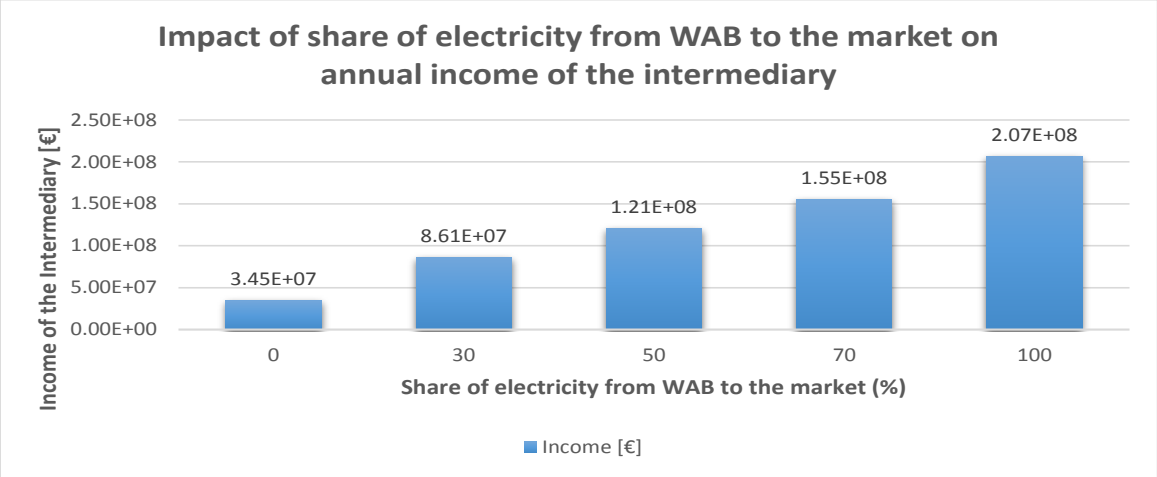

Figure 5.5: Impact of share of electricity amount to the market on annual income of the intermediary.

Source: Author's illustration

To observe the effect of the annual income, for each individual execution the share of electricity to the market has only been altered with 0%, 30%, 50%, 70% and 100% of five different set of parameters. From figure 5.5 it is clear that the annual income is increased with the increase of the share of electricity to the market. That means they are proportional with each other. Theoretically the annual income should be increased with the increment of the share of electricity to the market. This experiment shows that the obtained results are what the user expected and match with theoretical calculation of the annual income of the intermediary. This experiment claims that the AMIRIS model can produce accurate output with different set of parameters using batch processing .

Figure 5.6 shows an abstract processing mechanism how the AMIRIS model generates

different set of output with a single set of parameters and `Monte Carlo` number in a single run.

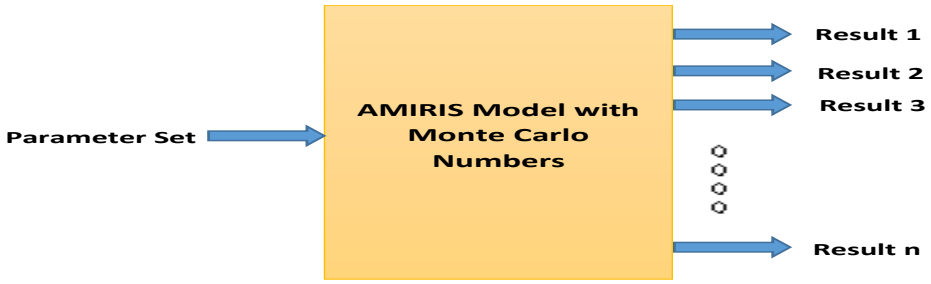

Figure 5.6: An abstract view of the AMIRIS model for generating different set of output with a single set of parameters and `Monte Carlo` number in a single run.

Source: Author's illustration

In order to produce different set of output with a single set of parameters in a single run, the user defines the number of executions and use `Monte Carlo` number inside the AMIRIS model. Hence the AMIRIS model will generate a new set of output in each execution. Gaussian distribution with a random seed has been used in old AMIRIS model to forecast the electricity production of the next hour for the power plant operators. The value of the random seed was set to 1. Because of the fixed value of the random seed, the AMIRIS model always produced same forecast of the electricity production for every individual execution. By using batch processing mechanism, the AMIRIS model executes 500 times to observe previous Gaussian distribution with the random seed which was set to 1. Figure 5.7 represents the forecasting of the electricity generation by Wind power plant for the first hour of the year 2013.

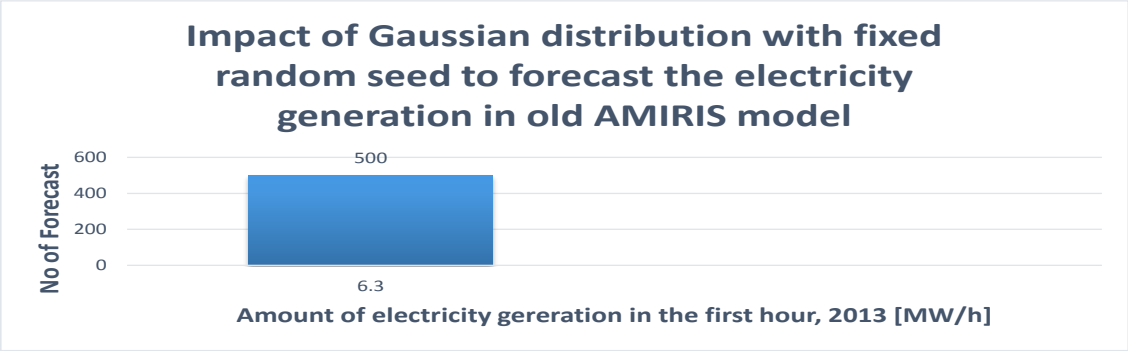

Figure 5.7: Impact of Gaussian distribution with fixed random seed to forecast the electricity generation in old AMIRIS model.

Source: Author's illustration

Although dynamic batch processing has been developed for maintaining multiple individual executions of the AMIRIS model in a single run, the model produces same output which does not meet its development goal.

In order to overcome this problem, an alternative approach has been researched and investigated. Currently Gaussian distribution with `Monte Carlo` number is used in the AMIRIS model. So Gaussian distribution randomly generates different values in each execution. The generated value is used to forecast the electricity production and the AMIRIS model can produce different output for each individual execution. Figure 5.8 demonstrates the impact of `Monte Carlo` number inside the AMIRIS model for predicting the electricity production by Wind power plant for the first hour of the year 2013.

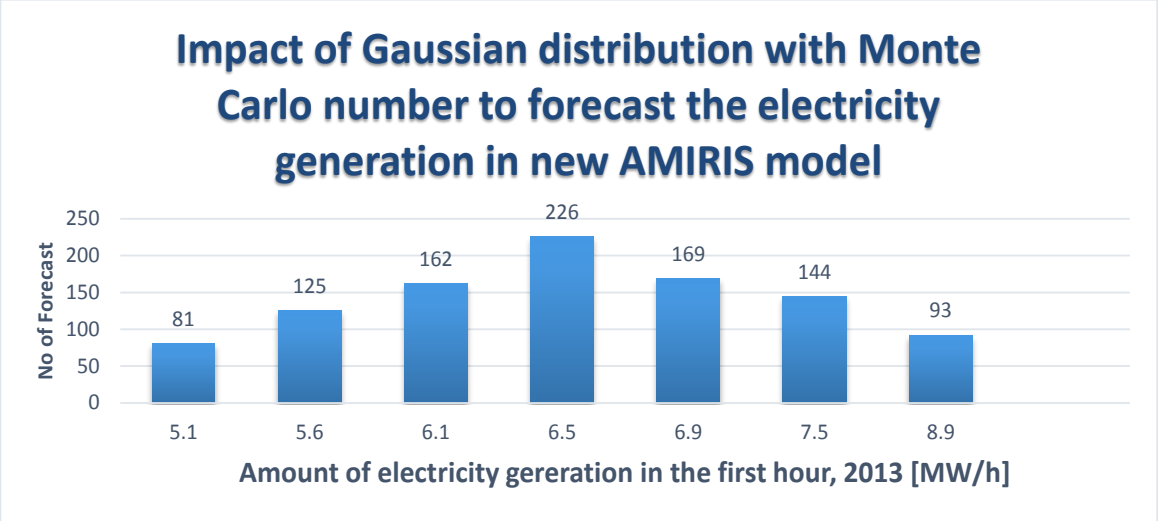

Figure 5.8: Impact of Gaussian distribution with variable Monte Carlo number to forecast the electricity generation in new AMIRIS model.

Source: Author's illustration

To observe the effect of `Monte Carlo` number into the AMIRIS model, the model has been executed 1000 times in a single run. The histogram measures the frequency of the forecasting of the electricity generation for Wind power plant with different ranges. This experiment demonstrates that now the AMIRIS model can generate different output with a single set of parameters using `Monte Carlo` number. Different set of generated outputs are very useful for statistical analysis which is one of the most important issues of the AMIRIS model.

Performance measurement

For the better analysis of the renewable energy market each year is divided into hours in

the AMIRIS model (one year equals 8760 hours). For each individual hour, every step of the AMIRIS model is executed. Previously the AMIRIS model was required to initialize the agents and define the model criteria for every run. An individual execution of the AMIRIS model was required approximately 0.46 minutes to simulate the entire process for a year. On the other hand, with the development of dynamic parameterization and batch processing, the AMIRIS model instantiates the model criteria at the beginning of simulation and the model criteria do not need to instantiate for every individual execution. Thus it saves the model initialization time. Currently the AMIRIS model takes only around 0.33 minutes for executing the whole process of a year in each execution. The number of executions is set with value 5 and the AMIRIS model takes in total 1.62 minutes to perform five individual executions in a single run. Hence newly developed AMIRIS model is 1.3 times faster than previous one.

In a word, development of a dynamic and flexible parameterization as well as batch processing mechanism has been significantly increased the performance of the AMIRIS model and expressively reduced human interaction to operate the model. Moreover, in a single run the model is capable to produce different set of output which used for better statistical analysis.

Chapter 6

Conclusion

The AMIRIS model is a powerful analytical simulation system for investigating and analyzing the market integration of renewable energy by the direct marketing. Previously the parameterization of the model criteria and of the agents was statically coded in the source code which is difficult to adopt with new parameters. Moreover, there was no batch processing mechanism to manage the recursive execution of the AMIRIS model in a single run. Several data storage applications, data validation mechanisms and XML parsers have been analyzed and researched to develop a dynamic and flexible parameterization mechanism. Furthermore, different batch processing mechanisms have been investigated for building a batch processing mechanism. Hence a dynamic and flexible parameterization mechanism has been built to automatically adopt the parameters for defining the model criteria and for initializing the agents of the AMIRIS model. In addition, batch processing has been developed for managing iterative execution of the AMIRIS model in a single run in order to get different set of results for better statistical analysis.

Both developed approaches have been experimented successfully on the AMIRIS model. The experiment shows that now the AMIRIS model can easily grasp new parameters automatically to define the model criteria and instantiate different agents as well as can manage the iterative executions in a single run.

In summary, development of dynamic parameterization and batch processing has increased the performance of the AMIRIS model approximately by 32%, has made the AMIRIS model flexible to adopt new parameters as well as has expressively minimized human interaction to operate the AMIRIS model.

Future Work

The AMIRIS model can be enhanced with parallel processing that can maintain paral-

lel executions of the model. By developing parallel processing the performance of the AMIRIS model can be increased significantly for iterative execution. The AMIRIS model can also be extended by cloud computing which can be really useful and convenient for remote execution of the model. Moreover, XML manipulation GUI can be developed to manage the parameters for defining the model criteria and instantiating the agents in a convenient way.

List of Figures

List of Tables

Bibliography

[BMU10] *Federal Ministry for the Environment, Nature Conversation and Nuclear Safety (BMU) and Federal Ministry of Economics and Technology (BMWi) (2010): Energy concept – for an environmentally sound, reliable and affordable energy supply.* 2010.

[Chi03] *Chira, C. 2003: Software Agents, IDIMS Report, 2/21/03, http://pan.nuigalway.ie/code/docs/agents.pdf.* 2003.

[Etz95] *Etzioni, O & Weld, D. S. 1995: Intelligent agents on the Internet: Fact, Fiction and Forecast. IEEE Expert, 10(4), 44-49.* 1995.

[Fli11] *Fligstein and McAdam (2011): "Toward a general theory of strategic action fields". Sociological Theory, Vol. 29, No. 1, pp. 1–26.* 2011.

[Fuc08] *Fuchs, Gerhard/Wassermann, Sandra 2008: Picking a Winner? Innovation in Photovoltaics and the Political Creation of Niche Markets. In: STI Studies 4(2), 93-113.* 2008.

[Fuc12] *Fuchs, Gerhard/Wassermann, Sandra 2012: From Niche to Mass Markets in High Technology: The Case of Photovoltaics in Germany. In: Bauer, Johannes/Lang, Achim/Schneider, Volker (eds.): Innovation Policy and Governance in High-Tech Industries: The Complexity of Coordination. Berlin: Springer, pp. 219-244.* 2012.

[Gen94] *Genesereth, M. and Ketchpel 1994: Software agents in Communications of the ACM, (July 1994) . ACM. 12,14, 48.* 1994.

[Goo93] *Goodwin, R. 1993: Formalizing properties of agents. Technical report, School*

of Computer Science, Carnegie-Mellon University, Pittsburgh. 1993.

[Has05] *Hasse, Raimund/Krücken, Georg 2005: Neo-Institutionalismus, 2. Aufl., Bielefeld: transcript.* 2005.

[Jac04] *Jacobsson, Staffan/Anna Bergek 2004: Transforming the energy sector: the evolution of technological systems in renewable energy technology. In: Jacob,Klaus/Binder, Manfred/Wieczorek, Anna (eds.): Governance for Industrial Transformation. Proceedings of the 2003 BerlinConference on the Human Dimensionsof Global Environmental Change, Environmental Policy Research Centre. Berlin, pp. 208-236.* 2004:.

[Jac06] *Jacobsson, Staffan/Lauber, Volkmar, 2006: The politics and policy of energy system transformation – Explaining the German diffusion of renewable energy technology. In: Energy Policy 34, pp. 256–276.* 2006.

[Jen98] *Jennings, N. R., Norman, T. J., & Faratin, P. 1998. ADEPT: An Agent-based Approach to Business Process Management. ACM SIGMOD Record, 27(4), 32-39.* 1998.

[LaM98] *Lange, D.B.; Mitsuru, O. 1998 .: Programming and Deploying Java Mobile Agents with Aglets,Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 1998.* 1998.

[LFP99] *Labrou, Y.; Finin, T.; Peng, Y. 1999: Agent Communication Languages: the Current Landscape.IEEE Intelligent Systems, 14(2): pp. 45–52, 1999.* 1999.

[Mil02] *Milborrow, David 2002: External costs and the real truth. In: Windpower Monthly 18(1), p. 32.* 2002.

[Ode00] *Odell, J. (2000). Agents: Technology and usage (Part 1). Executive Report 3(4).* 2000.

[Ohl08] *Ohlhorst, Dörte/Bruns, Elke/Schön, Susanne/Köppel, Johann 2008: Windenergieboom in Deutschland: eine Erfolgsstory, in: Bechberger, Mischa/Mez, Lutz/Sohre, Annika (eds.): Windenergie im Ländervergleich. Steuerungsimpulse, Akteure und technische Entwicklungen in Deutschland, Dänemark, Spanien und Großbritannien. Frankfurt a. M.: P. Lang, pp. 3–60.* 2008.

[Rus95] *S. Russell and P. Norvig 1995: Artificial Intelligence: A Modern Approach. Prentice-Hall, 1995.* 1995.

[Sea69] *Searle, J 1969: Speech Acts, Cambridge, MA, Cambridge University Press, 1969.* 1969.

[Syc98] *Sycara, K. P. 1998: Multi-agent Systems. AI ma gazine, Intelligent Agents Summer, 19(2), 79-92.* 1998.

[Tro09] *Troitzsch (2009): "Perspectives and Challenges of Agent-Based Simulation as a Tool for Economics and Other Social Sciences". Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Decker et al. (Edt.), Vol. 1, 10.-15.05.2009, Budapest, Ungarn.* 2009.

[Woo02] *Wooldridge, Michael 2002: Introduction to MultiAgent Systems, 2002.* 2002.

Declaration

I declare that I have written this work independently and respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as image, drawings, and text passages) are used in this work, I declare that these materials are referenced as follows (i.e.: quote, source) and, whenever necessary, consents from the author to use such materials in my work were obtained.

Signature:
Stuttgart, 13.01.2014