Chapter 1: Introduction

The University of Sheffield

Computer Science Department

Reusable Model Design Expression Languages (ReMoDEL)

Eman Abdulhamead Alzaid

Under Supervision of Dr. Anthony Simons

MSc Advanced Computer Science

August 2007

This report is submitted in partial fulfilment of the requirement for the degree of MSc in Advanced Computer Science
Signed Declaration

All sentences or passages quoted in this dissertation from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this dissertation have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Name: EMAN ALZAID

Signature:

Date: 29th August 2007
Abstract

This dissertation reports on an investigation into workflow information capture and automatic program generation strategies for the larger ReMoDEL project (Reusable Model Design Expression Languages), proposed by Dr Anthony J H Simons. The ReMoDEL project aims to allow end-users to design high-level models of information systems, from which the executable program code will be generated correctly by construction; and it will be possible to customise and adapt systems at the model-level, regenerating the executable code from scratch, without invalidating any existing saved business data.

The current work reported here focuses on business workflow and different ways of capturing and executing this. Initially, a flowgraph model was developed to capture sequence, selection, parallel composition and repetition of business tasks. Automatic procedures were designed to check the syntax of the flowgraph (the physical model) for its sound logical properties and to transform this into a structure chart (the logical model) with single entry and exit points, through the use of graph grammars. The logical model was made executable, using a variant of the Command design pattern.

The conclusions find that, whereas the transformed logical model suits the generation of standalone applications, this model is not so suitable for generating essentially stateless web-based systems, which fail to preserve block scope. Instead, a strategy was developed for generating JSP web-pages, linked to a MySQL database server, directly from the flowgraph models. Flows correspond directly to links between web pages, which represent sets of tasks. Atomic tasks are represented as primitive operations on the database, using standard JSP and MySQL operations.
Chapter 1: Introduction

Acknowledgments

I dedicate this dissertation to my parents and my husband for their unlimited aid. Special thanks to my supervisor Dr. Anthony Simons for his support and guidance throughout the project.
# Contents

**Chapter 1: Introduction**  
1.1 Aims of the project 1  
1.2 Project Deliverables 1  
1.3 Structure of the Dissertation 1  
**Chapter 2: Literature Review** 3  
2.1 Domain specific language 3  
  2.1.1 Domain Specific Language Requirements 3  
  2.1.2 Domain Specific Language Identification 3  
  2.1.3 Implementing Domain Specific Languages 4  
  2.1.4 Meta-models in domain specific languages 4  
2.2 Business Workflow 6  
  2.2.1 Business Workflow Languages 6  
    2.2.1.1 UML Activity diagram 6  
    2.2.1.2 Business Process Diagram Notation 7  
    2.2.1.3 Discovery Method’s Task Flow Diagram 8  
  2.2.2 Workflow Language Requirements 8  
  2.2.3 Workflow Meta-models 9  
2.3 Graph Grammar for visual languages 13  
  2.3.1 Regular Expressions Grammar 13  
  2.3.2 Context-free Grammar 13  
  2.3.3 Context-sensitive Grammar 13  
2.4 Code Generator 14  
**Chapter 3: Requirements Analysis** 15  
  3.1 Project Requirements 15  
**Chapter 4: Design and Implementation** 17  
  4.1 System Architecture 17  
  4.2 System Components 17  
    4.2.1 Workflow Elements Repository 17  
    4.2.2 Workflow Physical Model 20  
      4.2.2.1 Concrete Syntax 20  
      4.2.2.2 Abstract Syntax 24  
    4.2.3 Workflow Logical Model 27  
    4.2.4 Workflow Generator 28  
      4.2.4.1 Physical Model to Logical Model transformation 28  
      4.2.4.2 Generating Applications from Workflow Models 29  
      4.2.4.3 Atomic Task Executer 31  
**Chapter 5: Testing** 33  
  5.1 Testing the Checking Algorithm 33  
    5.1.1 Testing General Rules 34  
    5.1.2 Testing Graph Rules 35  
    5.1.3 Test the Performance of checking algorithm 38  
  5.2 Testing the transformation algorithm 38  
  5.3 Testing Atomic Task Executer 41  
**Chapter 6: Results and discussion** 42  
  6.1 Loop Presentation in Physical Model 42  
  6.2 Checking Algorithm 43
Chapter 1: Introduction

6.3 Generating an Executable Workflow Model 43
Chapter 7: Conclusions 44
Chapter 1: Introduction

Recently, many commercial tools have been developed to automate application development. Providing end users who don’t have any programming skills with tools that helped them to specify their system is the idea of Program-it yourself. Domain specific language is the technical concept that such tools are based upon. Users should be provided with languages, which use concepts of the domain that are understandable by them.

1.1 Aims of the project

This project aims to develop a business language that supports business users to specify their system specification using notations that they are familiar with. Business users usually use workflow notation to describe the business work. Workflows present the order of business tasks, business data influence, resources and responsibilities. The business workflow notation has been chosen as a base for the language that has been developed by this project.

In order to define a workflow language, all concepts that are used to describe workflows need to be identified. The rules that specify the relations between language concepts also need to be specified. These rules form the language grammar. Grammar is used to decide whether a construction of related concepts is valid or invalid according to its rules.

Most of the existing workflow notations use symbols to present its concepts e.g. UML Activity Diagram and Discovery Method Task Flows Diagram. Therefore, the target language that has been developed by this project is a visual language. To define any visual language, a special construct needs to be defined for each symbol and a graph grammar needs to be specified. A graph grammar is special type of grammar that is usually used to define visual languages rules.

1.2 Project Deliverables

The project develops two models for business workflow. The difference between the two models is in the concepts that are used to describe the workflows. The first model describes workflows as a group of related physical objects, so it’s called physical model. The second model presents workflow based on its logical structure, so it is called logical model. In other words, a physical workflow model uses business user concepts while the logical workflow model uses programmer concepts. The project, also, developed the physical model graph grammar that specifies whether a physical workflow diagram is valid or not.

However, the main challenging aspect of this project was in defining a transformation algorithm that converts a flow graph to its equivalent structured chart. The project, also, proves that structured charts are suitable to generate stand alone applications while using flow graphs is more convenience to generate web based applications.
Chapter 1: Introduction

An atomic tasks’ executer, also have been developed to generate executable primitive operations on a database.

1.3 Structure of the Report

This project outlines some aspects about domain specific languages, business workflows, graph grammar and code generator in Chapter 2, project requirements in chapter 3. Chapter 4 describes the design and implementation using class diagrams. Chapter 5 outlines the test cases that used to prove that the system algorithms are works properly. Chapter 6 presents the challenges that have been encountered and gives suggestions as a further work. The last chapter concludes with the project achievements and further work.
2.1 Domain Specific Language

Domain specific language (DSL) as declared in [1] is “a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain”. DSL is not a new idea. FORTRAN and COBOL are DSL’s where numeric arithmetic is the domain for the former and business for the latter. SQL, BNF, and HTML, also, are examples for commonly used domain specific languages.

However, DSL’s has many advantages over GPL’s [2]: DSL’s provide the domain specific notation and construction from the start while GPL’s need to be combined with application library; it is easier to analyse, verify, transform, optimise, and paralyze DSL constructs than in form of GPL; GPL’s must be executable but DSL’s don’t need to be. Combining a code generator with a DSL makes it an executable language and improves the speed of the development process by five to ten times than writing code manually [3]. DSL has, also, many advantages in increasing productivity, reusability, and testability.

2.1.1 Domain specific language requirements:

The requirements for the domain specific languages have been discussed by [8]. They categorise them as essential and additional requirements. Essential requirements are: (1) *Conformity:* the language constructs needs to present the main domain concepts; (2) *Orthogonality:* each construct represent one concept only; (3) *Supportability:* provide supporting tools for DSL; (4) *Integrability or Extensibility:* using the DSL with other languages or extending it by adding new constructs; (5) *Longevity:* making sure that the DSL supports its user for long time; (6) *Simplicity:* the language which should be stable and suitable for its users; (7) *Quality:* the language should include the general system quality conditions (ex: security) . The additional requirements are *Language Scalability:* the DSL is able to support constructs to develop large scale systems, and *Language Usability* which deals with the requirements such as accessibility and space economy.

2.1.2 Domain Specific Language Identification

According to [4], three primary features must be defined for any language: (1) *concrete syntax* which is the language presentation either textual or graphical; (2) *abstract syntax* that defines the language concepts, associations between them and the grammar to specify these associations; (3) *semantics* which deals with the meaning of the language models and programs.

Four ways were mentioned by [5] to identify language constructs: (1) *domain expert’s or developer’s concepts method* which is the process of defining languages that are assumed easier than other ways[5]; (2) *generation output method* (e.g. the languages
that are generated by using coding concepts like UML; (3) *look and feel of the system build method* which use the same concepts that are used by end user to model the constructs, so this method is assumed as the best approach; (4) *setting variability space method* which is assumed as the most difficult approach because it needs to predict the future variants.

### 2.1.3 Implementing Domain Specific Languages

For implementing DSL the best implementation pattern needs to be chosen. The writers of [2] have identified seven implementation patterns: (1) Interpreter by using the standard fetch-decode-execute to interpret the DSL constructs; (2) Embedding in a general purpose language: in this way the DSL constructs are embedded in the base language by using subroutines. This way supports using the base language features; (3) Compiler (application generator) in which the DSL constructs are converted into base language constructs and library calls; (4) Extending the general purpose language’s compiler or interpreter by adding the domain specific language rules. However, the process of extending interpreters are easier than extending compilers; (5) Pre-processor in which the domain specific language constructs are converted into the base language constructs; using macros is an example of this type;(6) using commercial off the shelf tools in a specific domain;(7) Hybrid approach which combines two or more of the previous patterns.

### 2.1.4 Meta-models in domain specific languages

Meta-modelling techniques are popular in describing the rules (abstract syntax) of domain specific languages. This section will present existing tools that use meta-models to design domain specific languages.

- **Software Factories** [9] is a Microsoft method to automate software development process by using models, domain specific languages, and code generators. A software factory based on three ideas: (1) software factory schema which describes how a DSL can be used and how the DSL’s models can be converted into code or other models specify the software architecture and the relationships between components;(2) software factory template and an extensible development environment.

- **Model Driven Architecture (MDA)**, is an OMG standard, aims to separate the business logic models from the implementation models by using OMG standards (e.g.: UML, XML, MOF) [6].

  MDA distinguishes between types of models: Platform Independent Model (PIM) and Platform Specific Model (PSM). PIM in MDA are designed using UML notations. Before translate these models to the code form, OMG converts them to a PSM. The PSM, then, is translated to the code.

  MOF is a language for defining the structural specification of models but does not define the behaviour of it. OCL (Object Constraint Language) is a formal language that used to specify the rules of the UML meta-models.
Chapter 2: Literature Review

Query View Transformation (QVT) is, also, an OMG standard which specializes in the mapping between models. This standard is interested in the transformation between models; storing models and converting them to text or converting text to model is out of QVT scope. QVT has assumed not convenient to define the behaviour of meta-models yet because OMG is still developing these standards [16].

XMF Mosaic [7] is a commercial tool for designing DSLs using meta-modelling technologies. In [7], Meta Object Facility (MOF) limitations to defining domain specific languages are addressed. The limitations are: (1) It does not support a platform independent for defining some semantic concepts like business process execution; (2) MOF uses XML to define the concrete syntax which makes it less readable; (3) MOF does not define abstractions for specifying user interfaces and tools in generic way [7].

To avoid the MOF’s limitations, a new approach has been developed by [7]. It based on construction of a layered eXecutable Metamodelling Facility (XMF). XMF consist of four components: (1) A virtual machine to support platform independent for the languages; (2) executable metamodelling language that has the minimum language definitions capabilities; (3) Domain specific language that based on general purpose language and metamodeling languages such as user interface and mapping languages (4) support a link between metamodels and an external UI application.

MetaEdit+ is a tool for developing and using DSL [3]. It enables users to build their tools by specifying model concepts, concepts properties, symbols, and the rules between these concepts. MetaEdit+, also, supports developers with generator editor to help developer to build the code generator.

Graphical Editor Framework (GEF) and Eclipse Modelling Framework (EMF) are Eclipse tools projects that support the domain specific language developer. Graphical Editor Framework (GEF) supports building graphical editor, while Eclipse Modelling Framework (EMF) helps in building models. But DSL developer needs the two tools to work together, so, Eclipse proposes a new project Graphical Modelling Framework (GMF) to combine GEF and EMF.
2.2 Business Workflow

A workflow describes the order of a set of tasks performed by various agents according to a defined set of rules to achieve a business goal. Workflow modelling has important role in increasing the speed of software development. Workflow models are used by developers to describe the task's behaviour and structure. This section presents business workflow languages, workflow language requirements and two workflow meta-models.

2.2.1 Business Workflow Languages

There are many graphical notations that can be used to describe the behaviour of business task. Here are some examples of these graphical business process modelling languages: the UML activity diagram from OMG, Business Process Modeling Notation (BPMN) form Business Process Management Initiative (BPMI) and Discovery Method’s task modelling notation. This section will describe briefly about the previous notations.

2.2.1.1 UML Activity diagram

An Activity diagram is used to model workflows, the business and operational aspects of a system. An activity diagram is a special case of UML state diagram which is used to present the state machine.

An activity diagram presents the events and the actions that cause the object to be in the particular state. The main elements of activity diagram can be divided into actions and control nodes (figure 2.1). The general types of actions are activity, call behaviour action, accept event and send signal. The control nodes include initial node, activity final, flow final, decision, meagre, fork and join.

![Activity Diagram Main Symbols](image)

Figure 2.1: Activity Diagram Main Symbols

The main disadvantage of UML activity diagrams, as stated in [18] is that there is no standard documentation that describes the complete syntax and semantics of the activity diagrams; specially the aspects related to the activity diagram itself not those which are inherited from the state diagram. The only way that can be used to describe the activity diagram features is through OCL (Object Constraint Language) which uses natural language to describe these features. Because of using natural language in defining activity diagram constraints, ambiguities may be in its semantics.
2.2.1.2 Business Process Diagram Notation

Business Process Modelling Notation (BPMN) has been developed by the Business Process Management Initiative (BPMI). May, 2004 was the first time that BPMN specification was published to the public [17].

The main aim of the BPMN was to define a notation that is easily understandable by all types of business users. To automate system development, BPMN works on building a mapping between business process design and process implementation.

Business Process Diagram (BPD) that defined by BPMN, is used a flowcharting technique for creating graphical models of business process operations. A Business Process Model is a group of graphical objects, which are activities (i.e., work) and the flow controls that define their order of performance.

The elements of BPD can be divided into four types: flow objects, connecting objects, swimlanes and artifacts (see figure 2.2). The flow objects include: (1) Events: events are divided into three types start, intermediate and end. Each event has cause and impact; (2) Activity which presents the concept task or work; (3) Gateway which is used to present the change in the control behaviour. In other words, it uses one symbol to show the change of the flow decision, merge, fork or join and internal signs to show the type of control. Connecting objects are used to connect diagram elements. There are three types of flows sequence flow, message flow and association.

Swimlanes includes two main types: pool which present a participant in a process; Lane which is a sub-partition inside the pool and used to organize and classify activities.

Artifacts include Data object which is connected to an activity that produced it, Group which includes analysing information and is used just for documentation without affecting on the diagram flows and finally, Annotation which is used includes comments to the reader.
Discovery Method’s Task Flow Diagram

A discovery method is a software engineering method that used for building object-oriented systems. Discovery method has its own model to present the order of business tasks. This model called Task Flow Diagram.

Task Flow Diagram uses the UML’s activity diagram notations for control nodes except that it uses an ellipse shape to present tasks and has its special way to present exceptions flows. A half diamond with a two paths: (1) The exceptional path which has a failure condition and ends by a failure; (2) The default path which is used if the failure condition is not true.

2.3 Task Flow Model Main Elements

2.2.2 Workflow Language Requirements

Workflow languages are used to define workflows. These languages should have constructs to satisfy requirements of the workflow systems definition. In [23], the writers presented the required aspects of workflow definition. These requirements is called in [] as complete requirement. The requirement includes: (1) the workflow language should support the description of workflow control structure; which includes decisions, loops, branching, and concurrent; (2) Support the definition of the role of human in task and recourses for example: database ; (3) provides different types of task definition: automated task which may be invoked by a workflow engine immediately or by external agents; user task which is based on the interaction between human (task performer) and the system to be accomplished; and manual task in which a human performs the task and the role of the workflow engine is in notifying task performer and accept a signal from the task performer as indicator of task completion. (4) Support the definition of data, decisions, looping and branching as an instances related to the process instance.

2.2.3 Workflow Meta-model

To define the structure and rules of a workflow model, meta-models are used frequently. In literature, many workflow models have been developed to present the different aspects of workflows. This section will present two models (1) A workflow meta-model that supports transformation; (2) A generic meta-model of a business process which captures a wide range of business workflow concepts [19].

![Task Flow Model Diagram]
2.2.3.1 Structured Workflow Meta-model

Eder[15] presents a workflow meta-model that supports workflow transformation. The model uses a hierarchical composition for the complex activity. The model presents a workflow as a sequence of activities. The Activities were divided into elementary and complex. Elementary activity can not be composed into basic type while complex activity composes of sequential, parallel, conditional or alternative sub activity. To identify the place of an activity, activity occurrence association was used. Occurrence consists of all the predecessors and successors of the activity. Therefore, one activity can be used several times in the model by changing the occurrence attribute.

Figure 2.3: “The Eder’s workflow meta-model” [15]
2.2.3.1 Generic Workflow Meta-model

A generic workflow meta-model have been developed by [19] based on business process theory, workflow patterns and Workflow Management Coalition (WFMC). The aim of developing this model was to establishing a framework to evaluate existing Business Process Modelling Languages.

To describe meta-model elements of this model, five perspectives have been used (Figure 2.4 is the model that developed by [19]). These perspectives are: (1) Functional perspective which captures the process elements (2) Organisational perspective presents where and who process elements are accomplished by; (3) Behavioural perspective captures when process elements are performed; (4) Informational perspective presents the information elements that used or created by a process; (5) Business process context perspective which describes the business process characteristics e.g. Business goal.

![Figure 2.4: "Generic Workflow Meta-Model" [19]](image_url)
2.3 Graph Grammar for visual languages

A grammar can be defined as a set of rules that specify a construct from a list of terminals. A construct is classified as valid, if it satisfies the grammar’ rules. As using grammars in textual languages to recognize the language by the parser, a graph grammar is used in visual language, diagrammatic language, to define the language syntax.

Graph grammar has many applications in many areas [21]: model and program transformation, syntax and semantics of visual languages, visual modelling of behaviour and programming, modelling, meta-modelling, and model-driven architecture, software architectures and evolution, refactoring of programs and software systems; security policies.

Many types of grammar are used to define language rules: regular expression grammar, context-free grammar and context-sensitive grammar. This section will presents three different graph grammar types.

2.3.1 Regular Expressions Grammar

This type is the simplest way to describe the language rules. The formal definition for a regular expressions is “an algebraic notation for characterizing a set of strings”[22]. Therefore, the main application for regular expressions is in defining grammars for texts, strings. Searching in strings is an example of regular expressions’ applications.

2.3.2 Context-free Grammar

The context-free grammar is more complicated than the regular exceptions because it supporting the syntactic structure for the language but its parsing does not depend on the context.

2.3.3 Context-sensitive Grammar

The context-sensitive grammar is more complicated than the free-context because it is take the context in its account while parsing the validity of a statement, in textual language, or a graph in visual language. The next section present as an example a context-sensitive graph grammar, called reserved graph grammar, that developed by [20].

A Reserved Graph Grammar can be used to describe the syntax of diagrams using labelled graphs and graph rewriting rules. Labelled graphs means that all the connection points in the nodes should be specified. Graph rewriting rules, also called productions, are specifying all possible structures that can be formed by language.

A production has two graphs: left graph and right graph. The left graph is, usually, a reduction to the right graph. A production can be applied to a diagram in two ways Left-application and Right-application. Left-application means replace a sub-graph from the host diagram with the right graph of the production. The reverse procedure is used when Right-application is used.
2.4 Code Generators

A Code generator is a form of transformation which takes a source model as input and converts it to an executable code. In [14], model transformation approaches have been classified into two main categories model-to-code and model-to-model. Our interest here is model-to-code transformations.

According to [14], model-to-code category has been divided into two types visitor-based and template-based. Visitor-based-approach is a simple way that converts the source model immediately to the target code without using meta-models technology. This way is based on defining the types of target model elements by using predefined element types.

In template-based transformations, there is a template which consists of meta-code parts that is used to extract information from the source model and then perform code selection and iterative expansion.

It is important to point out the MetaEdit+ [3] experience with code generators. They, based on meta-models, store all information about the model. To map the model into the code, each symbol of the model has a special code that takes the values of the symbol as arguments. It is also, important to point to the most important feature that should be taken into account by a code generator developer in designing code generators. The good code generator should produce a complete code directly from the model without the need to convert the source model into intermediate model during code generation procedure [13].
Chapter 3: Requirements and Analysis

As stated before, the project aims to define an executable workflow language to be used by end business users. Figure 3.1 shows the system use case diagram. The following two sections describe the project’s primary and secondary requirements.

![Figure 3.1 System Use Case](image)

3.1 Project Requirements:

Project main requirement includes:

1. Building the concrete syntax of the language which specifies the concepts that are included in the flow graph. Therefore, a meta-model is needed to describe flow graph elements. This model will be called physical model. This model should satisfy the business language requirements (see chapter 2 section: Workflow Language Requirements); the model should include all elements that are needed to describe a business workflow.

   The main workflow elements that should be provided by this model are: (1) start element ;(2) end with success; (3) end with failure; (4) decision ; (5) loop (6) business task; (7) flow; (8) fork and join;(10) actors ;(9) business data.

2. Develop the abstract syntax, grammar, for the workflow physical model using sensitive-context grammar type (see section 2.3.3 for the definition of sensitive-context grammar). A checking algorithm that uses the grammar rules to decide whether a physical workflow is valid or not also need to be built. This algorithm should check the structure of the physical workflow diagram and detect any error related to its structure. There are many types of errors that result in an invalid workflow diagram. Missing core nodes like start node and
existence of an unconnected node are two examples of the errors that may be founded in workflow diagrams.

3- Building a workflow logical model which supports transformation. This model presents the logical structure of a business workflow, programmer view. The concepts that should be provided by this model are: sequence structure, alternative structure, repetition structure and concurrent structure.

4- Supporting the logical model with executor to make the developed language an executable language and generate executable applications.
Chapter 4: Design and Implementation

This chapter describes the aspects that are related to project design.

4.1 System Architecture

The system consists of six main components: workflow physical workflow editor, workflow physical model, workflow logical model, and workflow elements repository, workflow generator and workflow web repository. Figure 4.1 shows the system architecture.

![Figure 4.1 System Main Components](image)

4.2 System Components

This section describes the components in details. UML diagrams have been used to show the relations between the components.

4.2.1 Workflow Elements Repository

The Workflow elements repository has been developed to include all the elements that are used by both physical and logical model. This includes constructs for the abstract business task and all aspects related to it. Business task aspects include data, actor and resource. In addition to the business task stuff, the repository also includes an important element that has effects on the flows of the workflow. This element, called test, is used in defining alternative paths and loops.

Figure 4.2 shows the two types of test: (1) external test which presents the user choice; (2) internal test which based on internal data.
In this repository a business task, called a basic task, is presented as a sequence of atomic tasks. An atomic task is a primitive operation on a database. The four atomic tasks that are defined by this project are: (1) Insert a new record; (2) Delete a record; (3) Update a record; (4) Retrieve a record. Figure 4.3 shows (a) the basic task association diagram and the inheritance diagram for atomic tasks in (b).

Because a workflow may have more than one resource, the basic task is designed to include a global resource list. The resource list includes all resources that are used by the atomic tasks. It is not necessary that all atomic tasks, which belong to the same basic task, use the same resource.
Chapter 4: Design and Implementation

Figure 4.4 Atomic Task Inheritance Diagram

Each atomic task has an execute method which executes a database query. Before using the execute method, one should make sure that all attributes have been specified. For example, to call the execute method of add new record, the database and table name need to be specified first.

For presenting a business resource, the project defines only one resource type which is a database. Figure 4.5 shows how the database class and database table class are designed.
4.2.2 Workflow Physical Model

The aim of this model is to present the workflow concepts and the relations between them, in other words, the concrete syntax and the abstract syntax for the flow graph. This section will be divided into two sections concrete syntax and abstract syntax.

4.2.2.1 Concrete Syntax

Because of the fact that workflow languages are visual languages, the concrete syntax of the developed language should include a construct for each visual element. That means defining a construct for each identical node, for each type of flows, and finally for the flow graph, the physical workflow.

A node may be one of the following a start, success-end, fail-end, task, decision, end-decision, fork, join, repetition and end-repetition. Figure 4.6 shows the inheritance diagram for the nodes.
Flows are used to connect between nodes. The model has three different types to classify flows: (1) Simple flow is the normal flow which just used to connect two nodes; has only one *from* and *to* attributes (2) conditional flow which is a simple flow with *test* attribute, see the section 4.2.1 for *test* definition. This type of flows is used only to connect decision node as a source to other node; (3) exceptional flow is a conditional flow but with two different destinations one for the normal path and the other for the exceptional path which ends with failure; this idea is the simulation for the way that Discovery Method’s task Flow Diagram present exceptions. Figure 4.7 shows the inheritance class diagram for flows.

The most important element is the one which used to present flow graphs; flow charts or what is called in this project physical workflow. A special construct is needed to present a physical workflow. This construct helps to improve the program readability because one of the project requirements is to transform the flow graph into structured chart, so the flow graph elements need to be send to that method. Therefore, instead of sending these elements separately it is better to encapsulate them in one object. The checking algorithm and some part of generator also process the flow graph, so using special construct will actually reduce program complexity. The physical diagram is not just used to store nodes and elements; it also includes any resource and data that used in defining workflow tasks. This will be useful for improving reusability; for example if a new task uses a resource which was used by a task that had defined before the new task, so no need to redefine the resource again. Figure 4.8 shows the association diagram for the physical-Workflow.
How Nodes and Flows are connected
This section describes the way that used to connects nodes and flows.

The principle that used to connect the nodes and flows is each node keeps track of the flows that are connected to it and keep track of the nodes that are connected by this flow.

Each flow has a source and destination, the project use two attributes to present them from and to. These attributes used to refer to the nodes that are connected by this flow.

A node, also, keeps tracks of the flows that used to connect it with other nodes. Each node has at least one connection point. Four types of connection points have been used: in-connection, multi-in-connection, out-connection and multi-out-connection. What a connection point is doing is keeping track of the flow or flows that connected to the node. Multi-in-connection and multi-out-connection may refer to more than one flow to refer to while in-connection and out-connection refers to only one flow. The attributes that are used by nodes to refer to flows are called in and out.

The type of node connection port, in or out, is based on the node type. Start and end nodes are the only elements that have only one connection point to keep track of just one flow. The out-connector type is used in the start node and the in-connector type in end node. All other elements have two connection attributes in and out. Decision has an in-connector and multi-out; the multi-out refers to one or more conditional flow. End- Decision has multi-in and out-connection. The fork and join has the same type of connection as Decision and End-Decision respectively except that fork’s multi-out-connection refer to normal flows instead of conditional flows. Repetition, End-Repetition and Task nodes have in-connection and out-connection.
Chapter 4: Design and Implementation

To describe how nodes are connected, assume there are two nodes $a$ and $b$ and a flow $f$ connects them as $a$ is the source and $b$ is the destination. So, the $out$ attribute of $a$ should refer to $f$ while the $from$ attribute of $f$ refers to $a$. On the other side, the $to$ attribute of $f$ refers to $b$ and the $in$ attribute of $b$ refers to $f$.

**How to Implement Concrete Syntax**

This section includes descriptions for all classes that used to present nodes, flows and physical workflow.

Class Node includes three attributes node id, node name and node type. Each of these attributes has its accessor and mutator method.

Class Repetition presents the start of loop structure. It has two connection ports in and out and has a repetition test attribute. Accessor and mutator methods are provided for the test attribute.

Class End Repetition presents end of loop structure. It has two connection ports in and out.

Class Fork presents the start of concurrent structure and has two connection ports in and multi-out.

Class Join presents end of a concurrent structure and has two connection ports multi-in and out.

Class Decision presents the start of the alternative structure with two connection ports in and multi-out.

Class End-Decision presents the end of the alternative structure with two connection ports multi-in and out.

Class Start presents the first point where the workflow starts from. This node has only an out port.

Class Task-Element presents the business task and has two connection ports in and out.

Class Success-End presents the point where the workflow ends successfully. This node has only one in port.

Class Fail-End presents the point where the workflow ends with exception. This node has only one in port.

Class Flow presents the simple flow which connects two nodes so, it has two attributes $from$ which is used to store a source node reference and to which is used to store a destination reference.

Class Conditional-Flow presents the flow which connects the decision node as a source to any other node. So, it has two attributes $from$ which is used to store a source Decision node reference and to which is used to store a destination reference. It also has a test attribute to store the value that usually used with flows out from decision.
Chapter 4: Design and Implementation

Class Exceptional-Flow presents the flow which connects a source node to two different distinctions; i.e. one from and two out one for the normal path and the other for the fail path called eto. It also has a test attribute to store the value that usually used with flows out from decision.

Class Physical-Workflow presents the flow graph as a group of related elements. It use different list for each type of nodes and flows. It also has a data list and resource list attributes to store all data that flows and resources that are used in the workflow. Moreover, it has a workflow name as attribute and its accessor and mutator methods.

4.2.2.2 Abstract Syntax
In order to check the correctness of the physical diagram structure, workflow rules need to be defined. To know what types of rules are needed, the errors that may occur in workflow structure need to be specified first. This section describes these errors and then presents the rules that have been defined to check the correctness of the validity of physical diagram structure.

The errors in a workflow diagram structure can be classified into categories:
1. Missing element errors: this includes missing core nodes like start or success end node or missing a control node e.g. if the diagram includes a fork element and no join element for that fork.
2. Connecting Errors: a diagram may include a flow which has one of its sides or both are not connected to other nodes. A node also, may be not connected properly e.g. missing to connect a node by flow from one side or from two sides if it has two sides.

The rules that check the validity of workflow diagram have been divided into general rules and structure-related rules. General rules concern checking element connections without caring of its place in the workflow diagram. On the other hand, structure-related rules determine the validity of workflow diagram structure.

General rules can be defined into:
1. Each workflow diagram has only one start point and one successful end point. only one flow comes out from start node and success end node also has only one flow reaching it.
2. Each task node T should be connected by two flows one from the previous node to T and the other from T to the next node.
3. Each flow should connect two different nodes, source and destination. A flow can not connect the node to its self.
4. A fork element should be a source for at least two flows.
5. The number of fork, decision and loop nodes should be identical to the number of join, end decision and end loop nodes respectively.

To define structure-related rules, a workflow graph grammar has been defined for the structured relations between nodes. A context-sensitive grammar, see section 2.3.3, has been defined to check the validity of a physical model structure. Table 4.1 shows the nine rules with its descriptions.
Chapter 4: Design and Implementation

**Rule 1:**
A valid workflow diagram should include one start point flowed by a task and one successful end point.

![Valid Workflow Diagram](image)

**Rule 2:**
A Task is a sequence of two tasks.

![Task Diagram](image)

**Rule 3:**
An exceptional flow may include a task before exception.

![Exception Flow Diagram](image)

**Rule 4:**
A Task may be followed by an exceptional flow which ends with exception.

![Task Exception Diagram](image)
**Rule 5:**  
A **concurrent structure** is a fork with an *n* alternative paths and Join element. \( n \geq 2 \)

![Diagram of concurrent structure](image)

**Rule 6:**  
A task may be a concurrent structure.

![Diagram of concurrent structure](image)

**Rule 7:**  
A **decision structure** is a decision with *n* alternative paths and end-decision element. \( n \geq 1 \)

![Diagram of decision structure](image)

**Rule 8:**  
A task may be a decision structure.

![Diagram of decision structure](image)
Rule 9:
A task may be a loop structure

<table>
<thead>
<tr>
<th>Task</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop in</td>
<td>out</td>
<td></td>
</tr>
</tbody>
</table>

| EndLoop in | out |

Table 4.1 Workflow Graph Grammar

**How to Check Physical Diagram**

In order to check the validity of the diagram structure the rules that stated in the previous section will be used. A diagram will be checked first using general rules. Then if the diagram is valid the graph rules will applied to it using Right-Application way (see section 2.3.3).

To check the validity of a diagram a parsing need to be used to reduce the graph according to the rules. Therefore, a Right-application is used when a production is applied to the graph.

Each graph rule was designed to search the diagram for any construct that has the same structure for its right side. If such construct is found then the graph rules will be applied by replacing its right side with a new construct which is same as the one on the left side of the rule. Here is the algorithm that is used to check the validity of the diagram structure. Figure 4.9 shows the checking algorithm.

```java
Boolean checkPhysicalDiagram( PhysicalModel hostDiagram)
{
    Boolean isValid = ApplyGeneralRules(hostDiagram)
    If ( isValid)
    {
        newHostDiagram=ApplyGraphRulesFrom_2_To_9(hostDiagram)
        While(newhostDiagram != hostDiagram)
        {
            hostDiagram=newhostDiagram
            newHostDiagram=ApplyGraphRulesFrom_2_To_9(hostDiagram)
        }
        If(firstGraphRule is Applied to newHostDiagram)
            Return true
    }
    Return false;
}
```

Figure 4.9 Checking Algorithm
4.2.3 Workflow Logical Model

The aim of this model is to present the same information found in a workflow as a logical structure chart. This model uses two main concepts to classify a workflow’s tasks: basic task and complex task. Logical model’s basic task has the same meaning as the task in the physical model but without the connection attributes. The complex task concept is used to refer to the control structure that used to co-ordinate other tasks.

The equivalent logical control structures to a workflow can be classified into four types: sequence structure, concurrent structure, alternative structure and loop structure. Each of these structures has its nature. Sequence structure is used to show the order of tasks; it arranges tasks in a queue first in first out. Concurrent structure shows the tasks that may be processed simultaneously; it is used to present the tasks between a fork and its corresponding join. Alternative structure is used to present the different flows of workflow that are based on conditions; it is used to present the tasks between Decision and End Decision using the physical model concepts. The repetition structure is used to present the repeated tasks or task; the tasks that are occurred between repetition and end repetition.

The logical workflow diagram is presented in this model as a sequence of tasks, basic tasks or complex tasks. The Complex task also may be composed of other complex tasks of any type.

For testing purpose a LogicalModelDisplay class have been developed to parse a logical work and print it using jTree structure. A jTree is a java component that displays a set of hierarchical data as an outline.

4.2.4 Workflow Generator

This section describes the transformation algorithm, generating applications from workflows models and executing atomic tasks.

4.2.4.1 Physical Model to Logical Model transformation

to convert from model to model a mapping transformation algorithm needs to be defined to translate each element from the source model to the equivalent in the target model. This algorithm takes a well structured physical workflow instance as input and creates the equivalent logical model instance.

To transfer a physical model to a logical model a parsing algorithm needs to be defined. The goal of parsing is to know the structure of the physical model and to specify it using logical model concepts. This parsing algorithm is designed to parse the
diagram using a recursive technique. The recursive call for the parsing happens usually at control nodes. These nodes are: Decision, Fork, Repetition and Exceptional-Flow. Figure 4.11 includes the transformation algorithm.

```c
Logical Workflow Generate_Logical Workflow(Physical Workflow PW)
{
    Logical workflow LW;
    LW.taskSequence=parse(start flow, new sequenceTask);
    Return LW;
}
SequenceTask parse(Flow next flow, SequenceTask currentSequence)
{
    If next flow is exceptional flow
        Create alternative structure alt
        branch1=Parse(fail Path, new SequenceTask)
        Branch2=Parse(success Path)
        Add branch1 and branch2 to currentAlternativestructure
    Else
        If next node is task
            Add task to currentSequence
            Parse( getNextFlow())
        If next node is decision
            Create alternative structure
            For each conditional flow i
                branchi=Parse (conditionalFlowi,)
                add branchi to currentAlternativeStructure
            Add currentAlternativeStructure to currentSequence
            Parse(getNextFlow())
        If next node is fork
            Create concurrent structure
            For each concurrentFlowi
                Branchi=Parse (concurrentFlowi)
                Add branchi to currentConcurrent structure
            Add currentConcurrent structure to currentSequence
            Parse(getNextFlow())
        If next node is Repetion
            Create Repetition structure rep
            currentRepetition=Parse (getNextFlow())
            Add currentRepetition to currentSequence
            Parse(getNextFlow())
        If(next node is fail-end)
            Add fail end to currentSequence;
    Return currentSequence
}
```

Figure 4.11 Algorithm to convert physical to logical workflow

4.2.4.2 Generating Applications from Workflow Models

This section describes how making workflow models generate applications. Two types of applications can be generated: stand alone application and Web-based Application.

Generating Stand Alone Applications

From logical workflows a stand alone application is generated immediately. By using Command design pattern [24], each logical task is designed with an execute method.
See section 4.2.3 for the details of the logical tasks. A class called `GenerateStandAloneApplication` is designed with one method `generateStandAloneApplication`, which takes a LogicalWorkflow instance as a parameter and executes it.

**Generating Web Based Application**

Our experiments to generate a web based application from the logical structure chart showed that structure charts are not convenient to generate web based applications. Therefore, we tried to use the physical workflow diagram directly to generate web based applications.

To achieve that goal, a parsing algorithm was developed to generate the workflow views from a physical workflow instance which does not include any concurrent elements; forks and joins. Views are classified into two types: task view and choice view. Figure 4.12 shows the class diagrams of the views.

![Figure 4.12 Workflow Views](image)

The algorithm that generates views works by exploring the flowgraph, identifying unique task and choice nodes, turning these into web views and linking each view to its appropriate neighbours. To link two views, two aspects need to be defined: the current view name and the next view name. To specify that, the flows’ id attributes are used as the current view and next view names. The flow’s id that comes out of the node forms the next view name and the one that comes in forms the current view name. Figure 4.13 shows the algorithm that is used to generate these views.
Chapter 4: Design and Implementation

4.2.4.3 Atomic Task Executer
This section includes the description for the architecture that was used to design the atomic task executor web service and the technologies that are used.

Architecture
The Model View Controller pattern (MVC) [24] is the architecture that used to design the atomic task executor. MVC is a three-tier architecture in which the business logic are separated the data management and from the user interface. Using a three-tier architecture is better than using a two-tier architecture due to the increasing performance, flexibility, maintainability, and reusability for using a three-tier architecture. Therefore, a three-tier architecture is a popular choice for designing web services.

The main components of the MVC are: (1) the Model, which used to communicate with the data and business functions that process them; (2) the View, which is used to present the data to user; (3) the Controller, which selects a suitable view for the user by handling the user’s interactions and the outcome of the model actions. MVC has many

```
WorkflowViews views
WorkflowViews GenerateWorkflowViews(Physical Workflow PW)
{
    parse(startFlow)
    return views;
}
parse(Flow nextFlow)
{
    If next flow is exceptional flow
    Create new choiceView with two choices
    Add currentView
    parse( failPath)
    parse(successPath)
    else
    If next node is task
    create new Task view
    nextView= getNearestFlowid (currentTask)
    add currentView
    parse( getNextFlow())
    else
    If next node is decision
    Create new Choice view
    nextView=getFlowsIds(node)
    for each conditional flowi
        parse(conditionalFlowi)
    If next node is Repetion
    create new choiceView with two choices
    nextView=getFlowsIds(node)
    parse(getNextFlow())
    getNextView(nextView)
    add currentView
}
```

Figure 4.13 Generating Workflow Views Algorithm
Chapter 4: Design and Implementation

benefits: (1) it provides many views using the same model in our situation view for each task; (2) it is easy to add a new type of task by writing a new view, enhance extensibility; (3) the Model and View are independent which enhance maintainability and testability.

Java Server Pages (JSP)[10] technology is used to implement the view. JSP provides a way to create dynamic web content. Because JSP is a member of the java family, it enables the creation of web-based applications that are platform independent. However, the four web views that are used are: add new record, delete record, update record, and retrieve record.

To design the controller, the Java servlets technology was used, because of its portability and the ability to access to all Java and its APIs. Figure 4.12 shows the architecture of the atomic task executer.

![Figure 4.14 Atomic Task Executer Architecture](image)

Figure 4.14 Atomic Task Executer Architecture

Figure 4.15 presents the form that is used to send a request to the TaskExecuter. The user needs to specify the atomic task type and the database details; URL, user Name and Password.

![Figure 4.15 Atomic task Form](image)

After submitting the form, TaskExecuter will specify which type of view to generate, based on the task type. for example, If the task type is add new record a form will be displayed with a list of tables’ names. After choosing the table name, another form will displayed which includes the input text for each field in the selected table.
Chapter 5: Testing

Three aspects need to be tested: the physical diagram checking algorithm, the transformation algorithm from physical diagram to logical diagram and, testing the workflow generator.

Unit testing is the way that used to check the system functionality. In this type of testing each unit is tested separately from other parts. Actually, the advantage of isolating and testing units separately is to allow programmer to detect errors earlier before integrate it with other parts.

This section will describe the test cases that used to test the main system functionality.

5.1 Testing Physical Workflow checking algorithm

To test the checking algorithm two aspects need to be checked: general rules and graph rules. For testing purpose a code script have been written for a valid physical diagram, see figure 5.1, to be used in testing both general rules and graph rules.

5.1.2 Testing General Rules

This section describes nine test cases which were used in general rules testing. In order to test, a valid physical workflow was built, see figure 5.1. The strategy that has been used to test each rule $x$:

![Figure 5.1 A Valid Flow Graph](image-url)
Chapter 5: Testing

1- Introduce the error that rule $x$ is looking for to the diagram in figure 5.1. Introducing an error to a diagram means making it to be invalid. This accomplished by adding or deleting some code lines to the corresponding code script.

2- Apply the general rule $x$ that detect to error after the alternation.

3- Print the error statement using Java `System.out.println` ; the error message state the type of the error and the node id.

**Test case 1:**
**Aim:** test rule 1 which checks the start element.
**Modification:** Start node is not defined for the physical model diagram for the physical model presents in fig 5.1
**Error message:** Missing Element: Start element

**Test case 2:**
**Aim:** test rule 2 which checks the success end element.
**Modification:** Success-End node is not defined for the physical model diagram presents in fig 5.1
**Error message:** Missing Element: Success end element

**Test case 3:**
**Aim:** test rule 3 which checks the connection status of the normal flows.
**Modification:** the destination of the flow that connects between start node and decision node has been set to undefined.
**Error message:** Connection error: Flow f1

**Test case 4:**
**Aim:** test rule 4 which checks the connection status of nodes of type Fork.
**Modification:** the fork is not connected from its in port.
**Error Message:** Connection error: Fork fork

**Test case 5:**
**Aim:** test rule 5 which checks the connection status of nodes of type Decision.
**Modification:** the Decision is not connected from its in port.
**Error Message:** Connection error: Decision condition

**Test case 6:**
**Aim:** test rule 6 which checks the connection status of nodes of type Repetition.
**Modification:** the Loop is not connected from its in port.
**Error Message:** Connection error: Loop rep

**Test case 7:**
**Aim:** test rule 7 which checks the connection status of nodes of type End Repetition.
**Modification:** the end Loop is not connected from its in port.
**Error Message:** Connection error: End Loop endrep
Chapter 5: Testing

Test case 8:
Aim: test rule 8 which checks the connection status of nodes of type Join.
Modification: the join node is not connected from its out port.
Error Message: Connection error: Join join

Test case 9:
Aim: test rule 9 which checks the connection status of nodes of type end Decision.
Modification: the end decision node is not connected from its out port.
Error Message: Connection error: End Decision endCondition

5.1.2 Testing Graph Rules

The strategy that has been used to test graph rules is by applying graph rules independently to the physical diagram that shown in figure 5.1.
The steps that have been done for all test cases except the last two test cases are:
1- Applying rules one by one to the figure5.1 physical diagram.
2- To see the resulting diagram, it will be converted to the equivalent logical model. The conversion is to take the advantage of using the algorithm that was implemented to print logical workflows.

Test Rule 2:
After applying rule 2 which states any two followed basic tasks in the original diagram will be reduced into one basic task, tasks 2 and 3 reduced into one task.

Test Rule 9:
After applying rule 9 to the diagram resulted from the previous test case. Rule 9 will reduce any repetition node that is followed by only one task node then end-repetition node.
**Chapter 5: Testing**

**Test Rules 5 and 6:**
Rule 5 and 6 reduce any concurrent structure, which each of its paths includes only one basic task, into one task element.

**Test Rules 7 and 8:**
Rule 7 and 8 reduce any alternative structure, which each of its paths includes only one basic task, into one task element.
To test this rules on the same diagram rule 2 need to be applied first.

![Applying Graph Rules 5 and 6](image1)

![Applying Graph Rule 2 Again](image2)

![Applying Graph Rules 7 and 8](image3)

### Table 5.1 Graph Rules Test Cases

**5.1.3 Test the Performance of checking algorithm**

To test the checking algorithm, two invalid structured physical diagrams have been developed. The checking algorithm will be used here; the general rules and graph rules will be applied to check the validity of the diagram. The Java `System.out.println` has been used to monitor the dynamic running state of each rule.

The first diagram is the same as the one that shown at the beginning of this section figure 5.1. The change that was made to this diagram was exchanging the place of Join node with the End-Decision node.

**The resulted output for the diagram before alteration:**
Applying graph Rule 2: Reduce Two Followed Tasks
Applying graph Rules 5 and 6: Reduce Concurrent Structure
Applying graph Rule 2: Reduce Two Followed Tasks
Applying graph Rule 2: Reduce Two Followed Tasks
Applying graph rule 9: Reduce Repetition Structure
Applying graph Rule 2: Reduce Two Followed Tasks
Applying graph Rules 7 and 8: Reduce Decision Structure
Applying graph Rule 1: Reduce Workflow
Valid Diagram
The resulted output for the diagram after alteration:
Applying graph Rule 2: Reduce Two Followed Tasks
Applying graph rule 9: Reduce Repetition Structure
Applying graph Rule 2: Reduce Two Followed Tasks
Invalid Diagram

The second diagram consists of jump flows the physical diagram presents in figure 5.2. The first diagram (a) is a well structured when the second diagram (b) consist a jump flow which make it an invalid structured diagram. The jump flow is the flow that connects between the two nodes task4 and the last end-condition, the node before end.

The resulted output before alteration:
Applying graph Rules 7 and 8: Reduce Decision Structure
Applying graph Rule 2: Reduce Two Followed Tasks
Applying graph Rules 7 and 8: Reduce Decision Structure
Applying graph Rule 1: Reduce Workflow
Valid Diagram

The resulted output for the diagram after alteration:
Invalid Diagram

![Diagram](image)

a) Before Alteration  b) After Alteration
Figure 5.2 Valid and Invalid Flow Graph
5.2 Testing the transformation algorithm

The input that used to test this algorithm is an instance of a well structured physical workflow and the output is a jtree that present the equivalent logical diagram.

The physical workflow has been build manually because the physical workflow editor has not been developed yet. Instead of presenting the code that presents the physical workflow, we will present the flow graph diagram and the logical workflow for each test case. The logical workflow diagram will be presented as a jTree using the DisplayLogicalWorkflow Class (see Section 4.3.4).

Table 5.2 shows five test cases were used to test the transformation algorithm.
Chapter 5: Testing

**Test Case 3:**
Flow graph Diagram

![Flow graph Diagram](image)

**Test Case 4:**
Flow graph diagram

![Flow graph diagram](image)

**Logical Diagram**

![Logical Diagram](image)
**Test case 5**
Flow graph Diagram

![Flow graph Diagram](image)

### Logical Diagram

![Logical Diagram](image)

**Table 5.2 Transformation Algorithm’s Test Cases**

### 5.3 Testing the Atomic Tasks Executer

Four test cases were used to test the atomic task executor, one for each atomic task. For testing purpose, a MySQL data schema has been created to be used by these test cases. After an atomic task has been invoked, the status of the database was checked to see whether the atomic task executor works properly. The atomic task executor works correctly if the database url, user name and password are specified correctly.
Chapter 6: Project Discussion

This section discusses the critical aspects that arose during the system implementation stage. Some examples are provided to describe these aspects clearly. There are, also, further suggestions to develop the system, which were not developed because of the time limit of this project.

6.1 Loop Presentation in Physical Model

In designing the physical workflow model, two new constructs were introduced to present loops, Repetition and End-Repetition, instead of using the Decision construct that is normally used in workflow languages. Although it may be possible to develop a graph grammar that uses a decision node to present loops, the main reason behind introducing the repetition constructs was the problem that appears while designing the transformation algorithm. The issue was the need to distinguish between Once-OR-More loop structure and Alternative structure; both have one flow that connects to it from in port and two or more out ports in the former and two in the later. Figure 6.1 shows how workflow languages include decision constructs with three different types of structures.

![Figure 6.1 Using Decision in Alternative and Loop presentations](image1.png)

Alternative Structure
One-Or-More Loop Structure
Zero-Or-More Loop Structure

The problem gets complex when having a loop composition structure, a one-or-more loop structure that includes another loop structure of the same type and the start point for the two loops is the same node. Figure 6.2 shows an example of such a situation.

![Figure 6.2 Loop Structure composition](image2.png)


6.2 Checking Algorithm

The checking algorithm checks the syntactic aspects of the flow graph, to ensure that it has a well-formed structure. For simplicity, we assumed that all selections were external user choices. If internal choices are admitted, further checking would be needed.

Checking the influence of workflow data is as important as structure checking. It gets more important when a workflow physical diagram has branches on internal value. Also checking could insure that such values are defined before they used.

6.3 Generating an Executable Workflow Model

Generating an executable logical workflow model is one of the main requirements of this project. To achieve this goal, two ways were suggested: (1) building an executable model that generates a stand alone application; (2) developing an executable logical workflow model that generates a web-based application.

The first method was done as described in section 4.2.4 and proves that the transformed logical model is appropriate to generate stand alone applications, while the other way failed due to the stateless nature of the basic protocol http. The proposed structure chart (logical model) was designed in a way that the tasks in lower level abstraction are controlled by the upper level not vice verse. Figure 7.3 shows a structure chart for a workflow consisting of a sequence, containing an alternative structure with two branches, each including only a basic task, followed by two basic tasks. To show when the problem occurs see that neither task3 nor task4 knows which next task should be invoked.

While the logical model is not suitable to generate a web based application, an attempt has been done as described in (section 4.2.4.2) to generate a web based application from the physical diagram. This attempt shows that it is more convincing to use the flow graph to generate a web based application.

Developing a graphical editor for the current system is another area that needs to be addressed as a further work.
Chapter 7: Conclusions

7.1 Achievements
The main aim of this project is to create an executable domain specific language, whose domain is a business workflow. In any domain language three aspects need to be defined: the concrete syntax which is presented in the physical model, the abstract syntax which is presented by the graph grammar that used to check a physical model instance and the language semantics which is illustrated by the logical model.

The abstract business task is presented as a sequence of atomic tasks. The atomic task is a basic operation on database which may be one of the following: update a record, display a record, delete a record and add a new record. An atomic task executer has been build to execute atomic tasks using JSP and Java Servlet technologies.

To make the developed language executable, a logical model executor is built to generate stand alone applications. However, this project proves that the logical model is not appropriate to generate web-based applications while it is worthy to generate stand alone applications. Alternatively, it is more convenient to use the physical models directly to generate web based applications by generating user views and associations between them based on the flows.

7.2 Further work
There are many areas need to be developed as further work: building a graphical user interface, checking the workflow data influence, and improving the workflow web based generator to generate applications.
7.0 References:


[14] Czarnecki K. & Helsen S., Classification of Model Transformation Approaches, OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture 1


[17] Stephen A. White, IBM Corporation. Introduction to BPMN.


