Polsko-Japońska Wyższa Szkoła Technik Komputerowych

PRACA MAGISTERSKA

Metamodel and Workflow Management System for Object - Oriented Business Processes

Metamodel i system zarządzania przepływem pracy dla procesów biznesowych zorientowanych obiektowo

Student	Wojciech Bauman
Nr albumu	2901
Promotor	dr Andrzej Jodłowski
Specjalność	Inżynieria Oprogramowania i Baz Danych
Katedra	Systemów Informacyjnych
Data zatwierdzenia tematu	20.04.2006
Data zakończenia pracy	31.04.2007

Podpis promotora pracy	**Podpis kierownika katedry**
.....	.....

Abstract

This thesis proposes a new approach to workflows and business process management, where processes and activities are objects capable of infinite nesting. This new model, built around the concepts of Stack-Based Approach (SBA), involves inherently parallel execution, dynamic process changes at run time, error handling, and resource management. Throughout this work, the author proposes extension to the SBA object store, which would enable it to store information about workflows. Complementary to this augmentation is the Stack-Based Query Language enhancement, which proposes introducing several new keywords for creating and manipulating business processes. It consists of special operations that will be performed on process definitions, process instances, activity definitions, and activity instances. The presented approach is not bound to any existing notation, so processes may be depicted more flexibly, and without focusing on notation-specific aspects or constraints. Within this thesis, there has also been developed a prototype implementation of the proposed workflow handling mechanism, written in Java and using XML files for storage.

Streszczenie

Niniejsza praca magisterska proponuje nowe podejście do zarządzania przepływem pracy oraz procesami biznesowymi, gdzie zarówno procesy, jak i aktywności są obiektami, które można nieskończenie zagnieżdżać. Ten nowy model, oparty na założeniach podejścia stosowego (*ang. stack-based approach, SBA*), uwzględnia inherentnie równoległe wykonywanie procesów, dynamiczne zmiany procesów w czasie ich wykonywania, obsługę błędów, a także zarządzanie zasobami. W pracy tej, autor proponuje rozszerzenia dla składu obiektów, które umożliwiłyby przechowywanie informacji na temat procesów pracy. Prócz tego, zaproponowane jest także rozszerzenie dla języka SBQL, zakładające wprowadzenie nowych konstrukcji dla tworzenia oraz manipulowania procesami biznesowymi. Proponowane rozwiązanie nie jest w żaden sposób związane z żadną istniejącą notacją, tak więc rzeczywiste procesy pracy mogą być odwzorowane bardziej elastycznie oraz bez skupiania się na aspektach czy ograniczeniach właściwych dla danej notacji. W ramach niniejszej rozprawy, została także rozwinięta prototypowa implementacja zaproponowanego podejścia do zarządzania procesami biznesowymi, stworzona w języku Java oraz wykorzystująca pliki XML do przechowywania danych.

Contents

1 Introduction to business process management

Business process management is nowadays a popular notion throughout the IT field. Starting from the system developers, through designers, architects, and ending with business analysts or marketing staff. Everyone uses this term, but it is understood in many different ways. What is more, there are many workflow management systems, standards, ad-hoc vendor-dependent platforms, and solutions that evolved from the academic environment. Most of those solutions have common problems that have not yet been addressed by any standard or vendor-dependent implementation. These problems can be broken down into several categories: (1) parallel execution, (2) dynamic changes, (3) exception handling, and (4) resource management. All four mentioned categories will be considered and addressed throughout this thesis, as well as the prototype implementation. Those issues will be reflected within the developed Stack-Based Approach and Stack-Based Query Language enhancements. As for today, none of the existing SBA/SBQL proposed extensions has addressed those problems in a uniform and decent way.

First one, the lack of parallel process execution, is probably the most troublesome limitation. It fails to reflect reality, where many processes are carried out in parallel, and only a few of them need to be aligned in sequential order. This sort of parallelization also needs advanced synchronization and routing mechanisms that will be able to choose appropriate process paths according to data, events, exceptions, or preceding tasks' statuses.

Dynamic process changes is another issue that lacks decent solution in today's workflow platforms or models. The process, once instantiated and launched, cannot change its definition until the execution ends. This approach severely impairs flexibility, because again, processes that occur in real life, tend to change depending on variety of internal or external factors, and thus are far more variable than their computer-designed equivalents.

Sometimes, exceptional situations arise during execution of a process. This can be anything from runtime exception during execution of a computer program, through deadlocks between waiting processes, to exceptional situation that resulted from bad or insufficient data and evaluated by the workflow engine itself. Proper handling of exceptions is the key part of workflow processing and may be useful when, for example, process should be terminated after a critical error or some special compensation activities should be executed to leave the system in consistent state.

Resource management is another area that will be addressed. Human or non-human resources could be allocated in many ways, like according to availability, capabilities (i.e. age or experience in case of human resources) or roles. Sometimes, particular activities must be performed by specific persons - not just everyone that satisfied the criteria. This type of allocation should also be supported, without notable flexibility loss or notational overhead.

As it was stated above, solution developed within this thesis will address all those issues. The proposed object store extension will enable processes to be stored and handled as other objects, and the Stack-Based Query Language enhancements will deliver functionality to create, instantiate and manipulate workflow processes. Noteworthy is the fact that the proposed approach to business processes is not tightly bound to any existing notation, hence processes may be depicted with much greater degree of flexibility, and without focusing on notation-specific aspects or constraints. Also, the XML-based Java implementation will be developed as a prototype of this flexible, object-oriented business process management and resource management solution.

1.1 Explanation of terms and introduction to the subject

Business process Series of tasks and outcomes related to some business activity. Consists of subprocesses, activities and decisions. Often, business processes are drawn in order to visualize relationships

between tasks, resources, roles and actions to be carried out. Business process modeling is used to document, standardize and reengineer business processes.

Activity Part of a business process, usually connected with executing some particular task or procedure. Several activities may be aligned and connected with each other, making up a business process. Each activity can have resources and/or roles assigned to it. Activities of a business process can be executed manually or automatically (i.e. as a programming language routine or database procedure).

Workflow More specific definition of a business process - involves operations on documents, structures of tasks altogether with required performer or resource information and some synchronization and routing constructs like preconditions, postconditions, and error handling (compensation). Being not a strict information technology term, workflow in most cases relates to interactions between people and information systems.

Workflow pattern Specialized form of a design pattern, which in the field of software engineering is well known as a common solution to recurring problem. Workflow patterns refer to proven solutions in development of applications related to workflow management or business process management. Probably the best known collection of workflow patterns is contained in the paper [Aalst et al.].

Process manager Software module, also called workflow engine, responsible for instantiating, executing and controlling workflow processes. To control a process, by means of a process manager, is to choose appropriate routes, handle errors and invoke other modules related to workflow, like resource manager in order to allocate resources to tasks or tasks to performers.

Stack-Based Approach An approach to query languages considering them as a subclass of imperative programming languages, and thus considering all proven notions and methods from programming languages applicable to query languages. Stack based approach (SBA) to query languages has been invented by Kazimierz Subieta. As for now, several different implementations of SBA exist.

Stack-Based Query Language Query language which has been designed basing on the concepts and principles of Stack-Based Approach. SBQL introduces non-algebraic operators, such as selection, projection, join, and transitive closure. Those operators' semantics can be easily expressed using environmental stack (ENVS) and results stack (QRES).

1.2 State of the art in the workflow management systems

The term "workflow" has been present in the computer science since the early 1980s. Its domain is thought to evolve from several fields, which include business management, simulation, process modeling and planning. The first aim of workflows was to organize and control processes on which businesses were based. Another goal was to automate and optimize those processes. Workflow can also be described as a series of information-intensive activities involving human-machine interactions. Sometimes, the term workflow is confused with business process management (BPM). However, the latter has a broader scope and involves processes from the business analyst's or manager's point of view, while workflow is used more often in the technical context ([Baeyens]). In some cases, workflow may even relate to sequence of user interface screens presented to the user of a particular application. In 1993, the Workflow Management Coalition has been started as the first effort to standardize the workflow field. Right now it has over 300 member organizations from all over the world, including vendors, consultants

and academic organizations. The so called reference model that has been published by this coalition defines the interfaces between the workflow management system and other actors, such as external systems. The five interfaces, as shown in Fig. 1, are described in table 1.

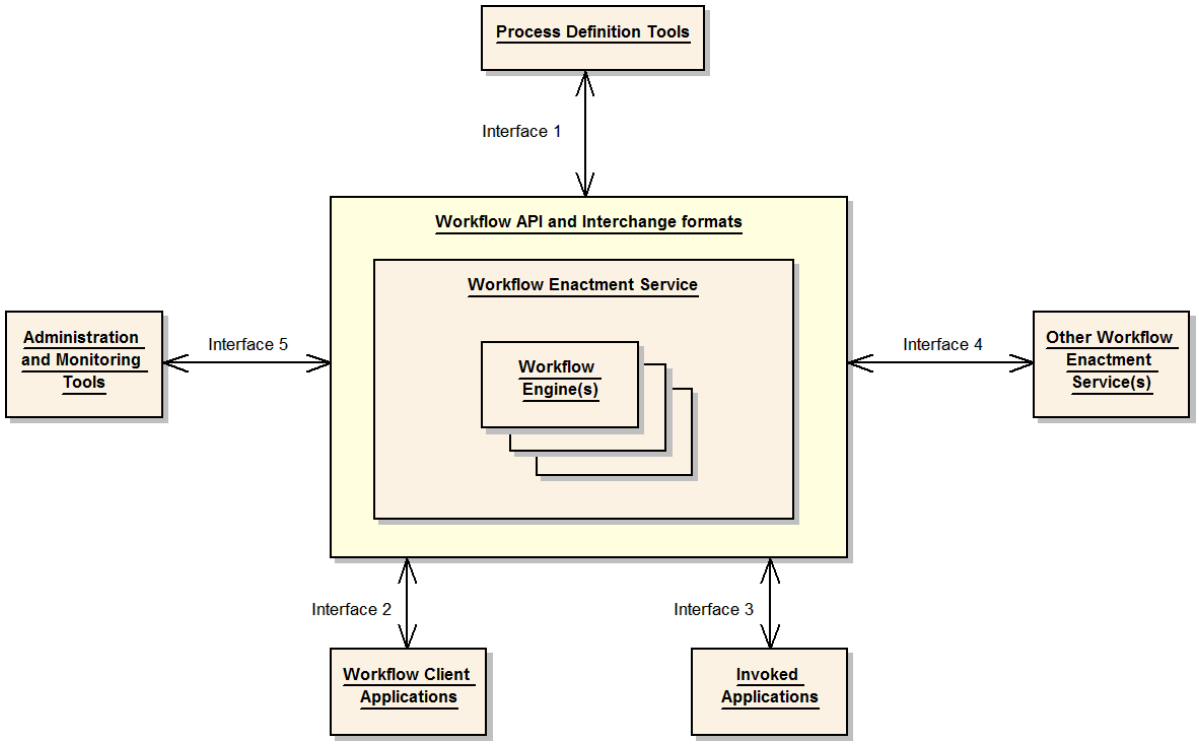

Figure 1: WfMC's workflow standard interfaces

Interface	**Function**
definition	functionality enabling process designers to create a process definition (for example using some graphics tool) and then deploy it to the workflow management system
client applications	the ability to invoke the system from the outside, in order to request service from the workflow engine, to control process and activity instances
invoked applications	enables invoking various external applications that are not the core part of the workflow system
interoperability	supports interoperability with other workflow systems, for example enabling to them to interchange data and work items
monitoring	functionality enabling monitoring and control over active process instances, as well as administration of the workflow management system

Table 1: Workflow standard interfaces

One of the specifications developed by WfMC is the XPDL (XML Process Definition Language),

which defines XML schema for declaring workflows. In 1996, the WfMC has created a glossary of terms with relationships between the key workflow terms (see Fig. 2, [WfMC]). As Tom Baeyens,

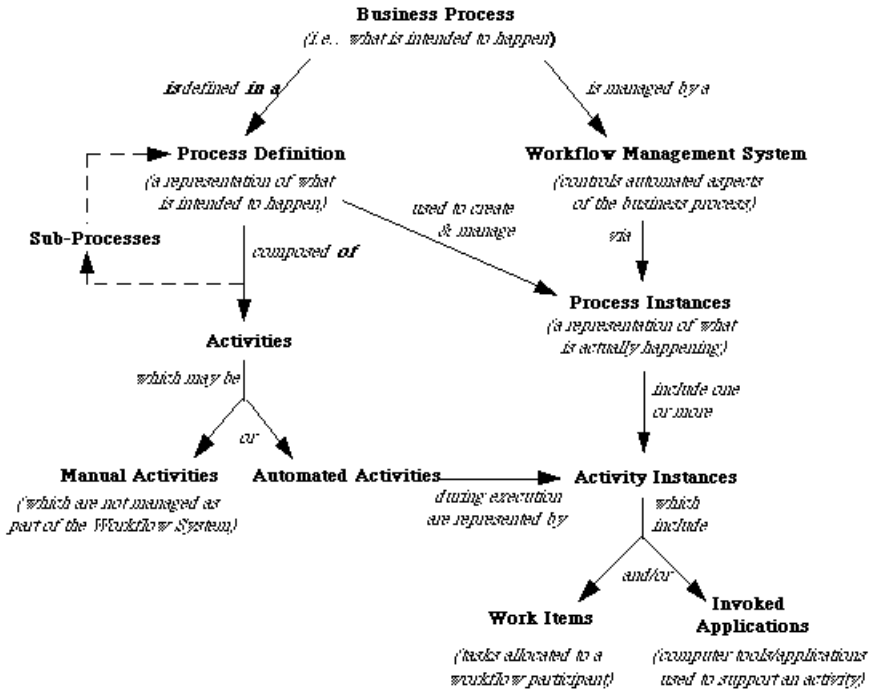

Figure 2: WfMC's workflow glossary of terms

lead architect of JBoss Inc. states in his article on workflows and BPM ([Baeyens]), today's workflow management systems are far less mature than relational database systems. This probably results from numerous different concepts for workflows developed individually. On the other hand, relational database system is a common, well established concept, widely understood among the community. This leads to the conclusion that in the near future, workflow management systems (further WFMS) are likely to develop heavily and gain much interest. Despite those predictions, however, even today WFMSs are intensely used in the field of enterprise application integration (EAI). Enterprise application deployments can consist of several heterogeneous systems, where each of them is focused on one clear purpose and is dedicated to support a single type of business processes (i.e. document processing, customer relationships management, supply chain management or enterprise resource planning). Application of a WFMS as an EAI platform in these scenarios would enable business processes to span over multiple dedicated applications, possibly without altering or re-deploying any of them. Flexibility of a WFMS lets designers define their own process definitions, instead of using ready-made and hard coded processes that are supplied with dedicated applications out of the box. It is believed by some people, that WFMS is the missing link in the enterprise application development. Those systems simplify application maintainability by centralizing business logic that would otherwise be scattered among database stored procedures, EJB components, and web applications, hence making systems developed this way far less maintainable and more cumbersome. Another option to use a workflow management system is to embed it into developed application. Used this way, the WFMS becomes another component of an application and is concealed from the end users. This approach is usually applied by companies that produce dedicated software systems, customized for a particular customer (eg. for national institutions or agencies). Among the most of the traditional process definition and modeling tools, as well as workflow

management systems, four layers can be identified and distinguished. Commonly understood notion of a process definition can be divided into those four pieces.

The **state layer** relates to expressing states and control flow, derived from standard programming languages, and Von Neumann architectures. States, which are also called wait-states, denote a dependency of a process upon the particular actor (such as human or computer system). WFMS then awaits any signal from that actor, in order to go on with the process. Control flow on the other hand, specifies execution paths of the process - including common constructs well known from the programming world, like loop statements, condition statements, and the like. When the processes are being executed, the need arises to keep track on the current state of the current process. This implied introduction of token, which corresponds to the notion of a program counter in Von Neumann architectures, and helps to visualize current process states and pass control flow to states that currently own a token.

The **context layer** involves data that is being attached to the process or particular activity during its execution. Depending on WFMS, variables can be stored as primitive types, objects, custom types or even references to external resources like filename in the filesystem or database record's identifier. Usually, those references are turned into meaningful information when they are to be presented or transfered to one of the heterogeneous systems that the WFMS links.

Another layer - the **programming logic layer** is the key part of the process' or activity's business logic. In this layer it is defined, what actions are to be made in reaction to which events. Some cases of programming logic may involve fetching data from a database, and other may involve performing an XSLT transformation or sending emails.

The last one - **user interface layer** is a way to transform data that has been input by the user (for example by using forms in a web application) into appropriate variables of a workflow process. Basing on that information, WFMS may decide whether to start a new process instance or how to route existing one.

1.3 Notations and theoretical foundations of workflow management systems

Workflow management systems have come a long way since 1980s. During their development, there were many attempts to establish a unified and universal notation for modeling and visualizing processes. One of the most widely used notation is the Petri net, whose enthusiasts maintain that it is the only formalized notation, with well defined theoretical foundations. Other notations for workflow processes include UML activity diagrams, Business Process Modeling Notation (BPMN) as a UML extension, and Pert diagrams. A relatively new approach to workflows has been developed as a concept related to the Stack Based Approach - it doesn't follow the foundations of Petri nets, and hence focuses on flexibility of process execution and enables dynamic changes.

Petri net

Petri net has been invented in 1962 by Carl Adam Petri in his PhD thesis. It is also known as P/T-net (place/transition net), because of two types of nodes: places (states) and transitions (transitions between the states). There are also directed arcs that connect places and transitions. Tokens in Petri nets indicate in which state is currently the control flow. Petri nets have gained much interest and have been applied in such areas as software design, data analysis, concurrent programming, and workflow management. In the latter case, Petri nets are considered as a foundation having strong theoretical background itself. However, the main drawback of applying Petri nets to workflow management is their limitedness resulting from Von Neumann architecture roots - all the activities and processes are carried out sequentially by default. What is more, this model decreases flexibility, as processes instantiated once have to stick to their initial definition until the end of execution. Despite their limited nature, Petri

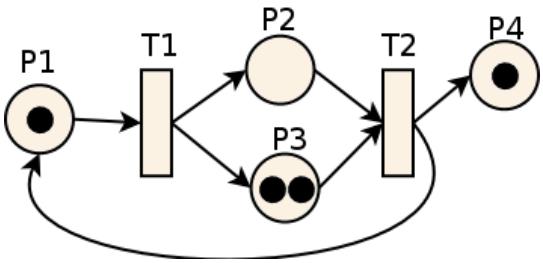

Figure 3: Example of a Petri net

nets or their derivative models have been widely implemented in various workflow management systems available on the market.

UML activity diagrams

Unified Modeling Language, or UML, is a standard language for modeling abstract schemes of information systems. Being a general-purpose language, it covers almost every aspect of modeling in the field of software engineering. UML has been defined officially by the Object Management Group (OMG), as a result of synthesis between Jim Rumbaugh's Object-modeling technique (OMT) and Grady Booch's method called, naturally enough, the Booch method. One of the most popular UML diagrams, used for depicting behavior in information systems, are the activity diagrams. They are usually used to express simple data flow or sequence of activities needed to carry out in order to achieve some goal. Sometimes, activity diagrams are used to draw workflow processes, but those are rather simple cases, that do not employ full-blown process management systems. For example, a general, high level view of a business process, or sequence of screens in a web applications can be expressed by using the UML activity diagram. Below is a simple activity diagram describing purchase process in a an online store.

BPMN as an UML extension

UML activity diagrams, as described above, are able to model some basic workflow process' behavior, but in case of more advanced and complicated constructs, process view depicted that way may fail to convey many important information, and thus making it farther from reality. Business Process Modeling Notation (BPMN) has been created to fill the gap between business processes' visualization and their actual behavior. It is a standardized graphical notation, developed by the Business Process Management Initiative (now part of the Object Management Group). Invention of this notation has been aimed at unifying process visualization - BPMN diagrams were designed to be understandable by system developers, as well as designers, architects, and even non-technical business users or analysts. It has been based on UML activity diagrams, but besides of several common elements, BPMN notation is far richer and focused on business processes. Elements that can be used to express processes can be divided into four major groups, which in turn consist of several element types. Structure of BPMN elements is presented in table 2.

Events are represented with circles, and denote that something has happened (1) before (start event), (2) during (intermediate event), or (3) after (end event) execution of a process.

Units of work, or **activities**, that need to be carried out during the execution of a workflow process are represented as rounded-corner rectangles. Activities can depict either indivisible, atomic tasks or nested processes. In the latter case, rectangle also has a plus sign at the bottom.

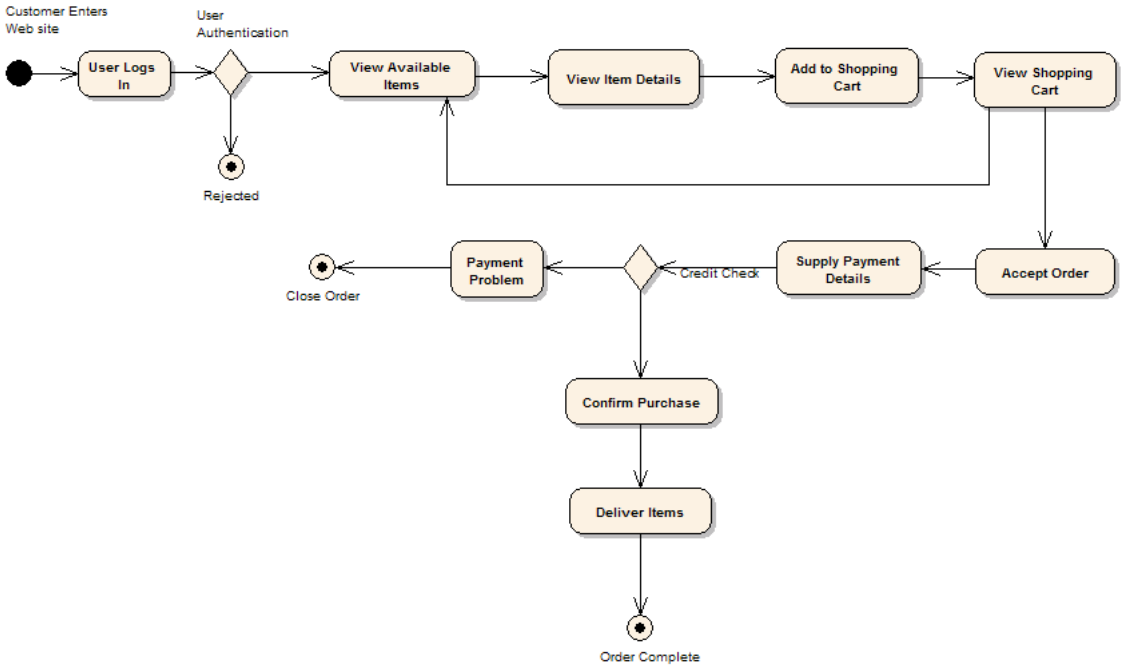

Figure 4: Sample activity diagram

Figure 5: Three kinds of BPMN events

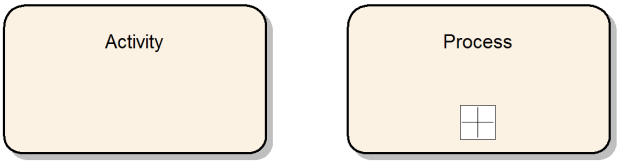

Figure 6: BPMN activity

Group	**Elements**
flow objects	events, activities, gateways
connecting objects	sequence flow, message flow, association
swimlanes	pool, lane
artifacts	

Table 2: Structure of BPMN elements

Elements used for routing between activities are in BPMN called **gateways**. A basic gateway determines different decisions, whereas the fork/join gateway depicts forking and joining of process paths. Third kind of gateway, the inclusive decision/merge gateway, is used to determine merging of routes.

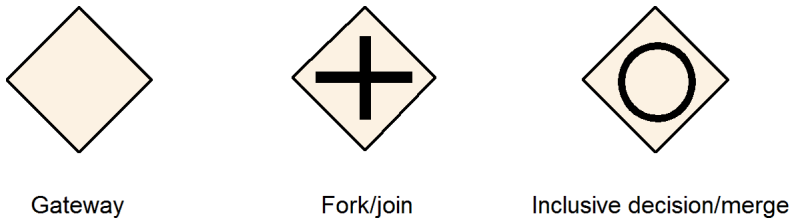

Figure 7: BPMN gateways

The BPMN notation has also three types of connectors that can be used between activities or between activity and gateway, etc. The **sequence flow** connector determines order of executing activities in the process, and it is illustrated with a simple arrow directed from source towards the destination. A diagonal slash across this line indicates default decision. **Message flow** connector specifies how asynchronous communication is done during the execution of a process. It is represented with a dashed line with an open arrowhead. Third connector, the **association** is used to associate data or artifact to a flow object, and it is visualized as a dotted line with no arrowhead.

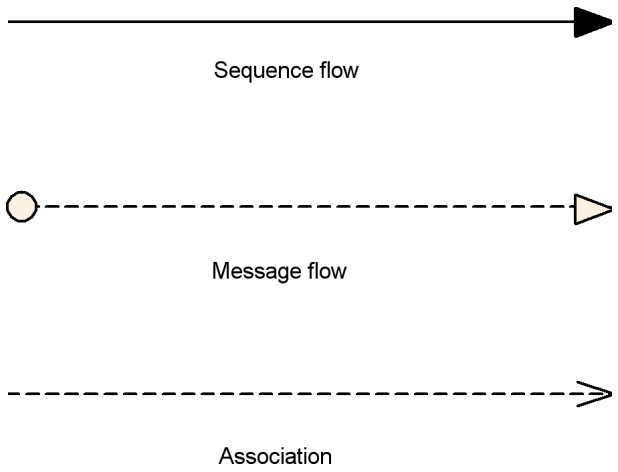

Figure 8: BPMN connections

Often there is a need to separate activities of a process that are executed on different systems or just

to logically organize those activities into categories. **Pool** and **lane**, generally termed **swimlanes**, are constructs that enable this categorization. They both look alike and can contain flow objects, connectors and artifacts. However there is a slight semantic difference between them - lanes are sub-parts of the pool, and thus one pool can contain many lanes.

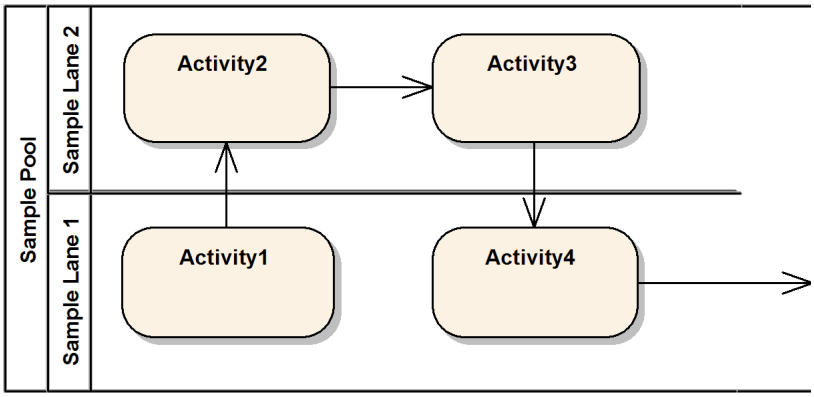

Figure 9: BPMN swimlanes

To increase readability of BPMN diagrams and bring more information to them, **artifacts** have been introduced. The three artifact types are: **data object** (visualizes data creation or flow along the process), **group** (used simply to group activities in the process), and **annotation** (boxes with text containing additional information about the diagram, that make it more understandable and self-explaining).

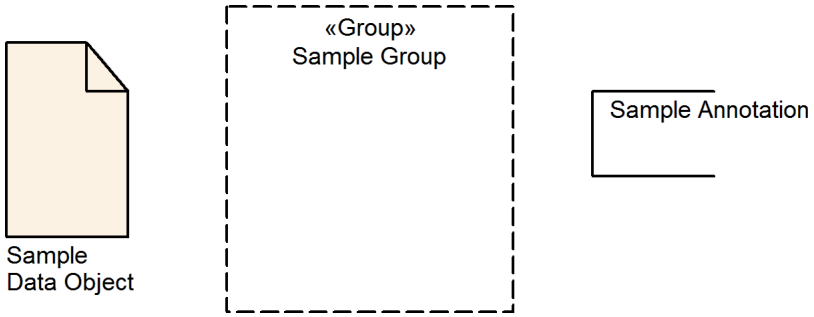

Figure 10: BPMN artifacts

PERT charts

The Program Evaluation and Review Technique is a model for project management that facilitates decision-making and is based on events and activities. It has been invented in 1958 by the United States Department of Defense. In PERT charts, emphasis is put mostly on time, not cost of the project. This diagram displays connected events as circles with numbers inside. Each of that circle represents a milestone. Events are connected with activities which, by convention, are drawn using arrows. Activities of a PERT chart can't be carried out as long as their nearest preceding activities haven't been finished. Sample PERT chart is presented below.

PERT diagrams are also used to visualize workflow processes. Sometimes they are even considered as a better alternative to Petri nets, because of their inherent ability to structure processes hierarchi-

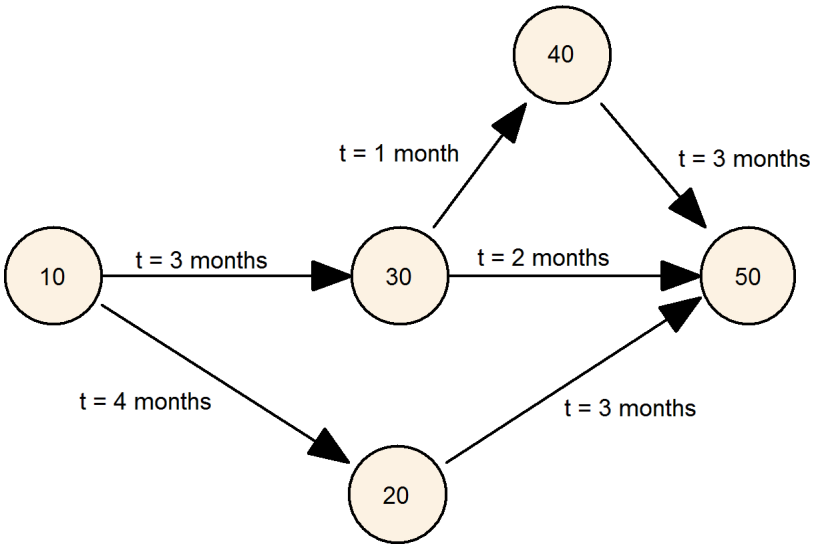

Figure 11: Sample PERT chart

cally and capability of infinite nesting of processes. These diagrams can also help in depicting more parallel processing, which, unlike in Petri nets, is not strictly following the Von Neumann architecture conventions.

The new approach to workflows

Apart from the standards proposed by OMG or WfMC, and various non-standard, vendor-dependent workflow solutions, the new vision of process management has been developed. It has been designed around the Stack-Based Approach to query languages and advocates more object-orientedness and parallelization in the field of workflows. A process, according to this approach, is a data structure with specific attributes and a procedure containing business logic that can be invoked during execution. Processes defined this way could contain any number of nested subprocesses, each of which would be inherently parallel. Such definitions would be capable of dynamic changes, according to current needs, and thus will not be limited to strict execution order defined at design time. This approach is much closer to reality than, for example, Petri nets, where, as mentioned in previous sections, execution path is firmly described and cannot be changed afterwards. PERT chart is considered a relevant visualization tool for workflow processes in this approach. It defines only which activities have to complete before another are started, but says nothing about the actual order of processing. Exact execution path is determined at runtime, using the pre- and postconditions contained in the process instances. Those conditions can not only establish routing between activities, but they can also act as synchronization blocks, and other constructs from the workflow field. Process management prototype developed within this thesis is based mainly on this new approach to workflows, as the author believes it to be powerful, flexible, and innovative solution to business process management.

1.4 Business process management solutions available on the market

Numerous workflow products are available nowadays. Some of them are open source systems, free to use even in the commercial environment. Probably the most popular open source workflow platform is **jBPM** from JBoss inc. While being a relatively new product on the market, it has gained much interest

in the community and is considered a stable and mature solution. Its current version supports two process languages: jPDL (Java Process Definition Language) - XML-based language for defining business processes, and BPEL (Business Process Execution Language) - language aimed to enable large-scale programming, which means long-running processes with asynchronous communication and integration via web services. jBPM, as its authors say, is a highly flexible and scalable process engine. It can run inside a J2EE container or as a standalone application. What is more, many processes, possibly defined in different languages, may execute simultaneously within the same workflow engine instance. It also integrates well with existing products, such as Jboss Seam (a Java Server Faces framework implementation) for handling page flow in web applications. Jboss jBPM is attractive not only to open source software enthusiasts, but it is also deployed in large-scale enterprise applications, replacing expensive commercial workflow engines.

Another business process solution from the open source group is Codehaus' **Werkflow**. This engine is not very popular, probably because of immature state of the code, performance problems and lack of serious customer support. Werkflow process model is based on Petri nets, and for defining processes it uses the Jelly programming language (an apparently failed attempt to create a programming language based entirely on XML). What is more, it lacks visual process editor, so all the process definitions must be written directly in XML. All in all, workflow engine from Codehaus seems to be rather a proof of concept than real solution that would be capable of solving real-world problems.

OSWorkflow from Open Symphony is yet another XML-based business process management system written in Java. While creators of this engine advocate writing process definitions directly to XML files or developing a custom GUI designer in-house, the latest version of OSWorkflow includes graphical process designer. The XML language used to define workflows does not conform to any of existing standards. Unlike the previously discussed solution, this one supports variety of external calls from within processes, like simple Java classes, classes retrieved via JNDI, and even remote EJBs or SOAP invocations. OSWorkflow is a low-level workflow solution, and its authors clearly emphasize it. Process definitions written in XML cannot be changed by everyone, i.e. business analysts, and are not intended to. Authors maintain that only the programmers are capable of making any changes to workflows.

Regardless of wide utilization of open-source workflow management software, most of the platforms applied today are quite expensive, enterprise workflow management engines produced by large companies. Those systems usually conform to existing standards (like BPEL, BPMN, etc.), and offer much more than simple workflow management engines. Integration with web services, ability to serve as a backbone for Enterprise Application Integration (EAI) solutions, and sophisticated GUI designer tool is a must for advanced commercial workflow systems deployed nowadays. Example of a full-blown workflow suite is IBM's **Websphere MQ Workflow**. For defining processes, it uses the FDL Workflow Definition Language which, as a notation is clear, comprehensible, and similar to BPMN. Another product from another big vendor is BEA's **Weblogic Integration**. Different vendor means different proprietary notation - this time it is Java Workflow Language (JWF). The JWF files are just plain Java classes with annotations describing routing logic, and also some detailed business logic that is referenced via XQuery or Java methods. The Oracle Application Server integration platform from Oracle is a set of products to build applications and integrate business processes across various, possibly heterogeneous, systems. One of the components of this suite is **Oracle Workflow**, a workflow engine. For defining processes, it uses its own proprietary standard - WFT files. Logical structure of business processes written according to this standard resembles Petri nets. Oracle Workflow can be embedded in user applications, providing support for task lists and routing in accordance with application events. This workflow management system also enables processes to perform external method calls - PLSQL database stored procedures or methods stored in Java classes.

2 Augmenting the Stack-Based Approach with workflow capabilities

Stack-Based Approach is a methodology for object-oriented query/programming languages. SBA encourages adapting techniques known from the programming languages to the world of database query languages. Therefore, this approach blurs the boundaries between those two sorts of languages, uniformly covering all their aspects. The Stack-Based Query Language (SBQL) adheres to this new approach. It has been designed from the practical, not theoretical point of view, hence it delivers more power to the users. Elements present in other programming or query languages, like syntactic sugar or mismatch between human and machine semantics, have been reduced in SBQL, resulting in relatively concise, clear and comprehensible syntax.

2.1 Stack Based Approach and Stack Based Query Language extensions

SBA and SBQL are highly extensible, so many projects, especially in an academic field, involve numerous extensions. These include updatable object views, dynamic object roles, optimizations, and so forth. However, so far it hasn't been extended with structures or keywords that would enable business process definition, storage, and manipulation. Therefore, this problem has been undertaken within this work. The introduced object store and language elements constitute a consistent and thorough solution to numerous problems related to workflows. Example process described below will be used to show some of those issues and their resolution based on this new approach to business process management.

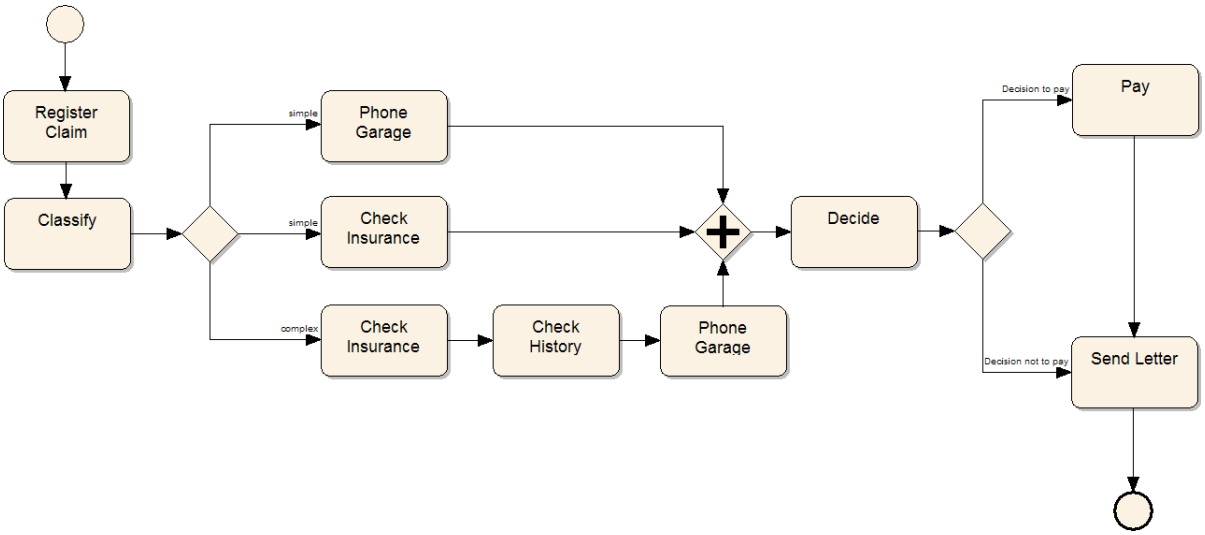

Figure 12: Insurance claims handling process

This process reflects insurance claims handling in an insurance company. For simplicity, we will narrow this company's activity to handling claims resulting from traffic accidents. Employees of this company register all incoming claims. After the registration is completed, a clerk analyzes particular cases and classifies them as simple or complex. Further process path depends on the claim type. Thus, for simple claims, insurance should be checked and an employee should phone garage. Those activities are independent and could be carried out in parallel. For complex claims, a clerk should check insurance, check car's damage history, and then phone garage. Activities of the latter claim type, depend on each other and should be attained sequentially. After executing activities in both of these scenarios, a decision is made. In accordance to this decision, money will be payed or not. Regardless of the outcome, the

insurance company has to send a letter to the claim originator. This is a simple example, yet enables demonstration of some vital features of this workflow SBA/SBQL extension.

Probably the most crucial and recurring problem of workflows is lack of inherent parallel execution of processes. This new approach assumes every process to be an object that is capable of independent and parallel execution. High level synchronization mechanisms have been introduced, in order to enable specifying enactment order when such need arises. By default, the only determinant that influences process flow is the availability of resources. This flexible mechanism enables building a wide variety of routing rules, based on process context, data, events, or errors. In the example process shown above, most of activities need to be carried out sequentially, in specific order. However, there is still some room for parallelization - the last two activities do not have to be executed one after another. The "Pay" activity should look up data related to decision that has been made earlier (within the "Decision" task), and according to its value - execute or not. At the same time, an appropriate letter should be sent to the claim originator - there is no need to wait until the "Pay" task is finished, because the decision has already been made. Both the "Pay" and "Send Letter" activities should have defined preconditions, in accordance to which, further process path would be determined.

Another important matter, dynamic process changes during execution, is also troublesome to many workflow engines and lacks decent solution among today's business process management platforms. What if the company's policy changes and some new path for processing requests arises? According to the process definition established at design time, it is not possible to reflect such changes. However, due to this new approach to workflows, it is possible to alter process definition from within procedural section of any activity. This concept highly increases flexibility, as processes are capable of adapting to rapidly changing situations and therefore reflecting reality with bigger degree of fidelity.

Error and exception handling is another vital issue when it comes to process management systems. During process execution, just like in real life, something can always go wrong. By exceptional situation we mean anything from some low-level exceptions thrown from within executing code, or errors defined on the higher level of abstraction and meaningful from the business point of view. All of these situations should be handled appropriately, which would increase the chance of recovery after serious or minor failures. Workflow extensions for SBA and SBQL developed within this work introduce the compensation mechanism. Every activity during process execution can go wrong and that is when the compensation procedures may turn out to be very helpful in leaving system state intact. Let us consider the example process described above. During the "Decision" activity, an erroneous decision could be made, for example, not to pay money. This error could be detected just after the letter had been sent to the claim originator. Relevant compensation procedure for this scenario would include paying money to that customer, and sending another letter. Its contents would include statement about cancellation of the previous one.

Key part in business process handling is resource management. This problem arises to even higher rank, when process parallelization hinges only upon resources availability. In the insurance claim handling process, we would like to assign decision task to the higher employee, with more experience and special skills. Through this SBQL extension, we are able to specify which group of users should perform specific tasks (assignment by roles) and additionally, we can define some special conditions for capabilities of potential performers that have to be satisfied. In our example, we can say that the "Decide" task can only be carried out by users of group "managers", additionally having at least 10 years of experience in the field of insurance law. This flexible performer specification leverages SBQL queries as a natural way of fetching data in this environment. Resource management is not only limited to human resources, but utilizes also many others, just like in real life activities. They are also allocated using SBQL queries and procedures, which are executed at run time.

Subsequent sections of this chapter describe proposed elements that should be added to object store, as well as the SBQL language, in order to support handling of business processes.

2.2 SBA object store extension for storage of workflow processes

The proposed SBA object store extension consists of seven new types that are the foundation for this workflow extension: **activity definition**, **activity instance**, **process definition**, **process instance**, **attribute**, **resource**, and **resource definition**. Every object has its internal, not readable identifier, and some of them (activity definition, activity instance, process definition, and process instance) also have special flag indicating that particular object is of specific type. Each of them, together with properties or associated objects will be described below, together with a brief description of usage and applicability in the already introduced example business process. More detailed, technical description of those elements can be found in Appendix A.

Activity definition

Activity definition is the main building block for workflow processes. According to those objects, activities are instantiated and can constitute a process definition. Designers or developers using activity definitions have to specify what actual work needs to be done, who will perform activities of this instances and what kind and amount of resources will that execution consume.

name - a unique, externally readable and meaningful name of the activity definition. This property can contain business identifiers, as well as names by which the activity definitions could be identified or distinguished by the software developers. In our sample process, names of the activity definitions will be "ClassifyDefinition", "DecideDefinition" or "PhoneGarageDefinition".

description - description of the activity definition. Can contain detailed information about what particular activity definition is for and what it should do. For example: "Decide if the claim will be admitted and money will be paid".

work - procedural part of the activity definition that specifies what should be done within all instances of this activity. This SBQL procedure will be executed in case of an automatic activity, after preconditions have been checked and before nested activities of this activity are launched. This element constitutes an atomic work unit, but it can consist of many procedure calls nested hierarchically. This element could be used to, for example, perform some business logic, call external systems or make operations on the data bound to the activity or process. "Check Insurance" activity from our example process could, for example, query special database containing policies and according to the outcome, set relevant attribute of this activity instance.

performer - performer who will perform activities instantiated using this activity definition - may be specified in numerous ways, eg. depending on role, qualifications or other properties; this attribute is used as a definition for eligible performers - actual resource that will be assigned to perform particular task will be determined dynamically at runtime, using this query. Example queries that can be used to determine performers in the "Decide" activity, within insurance claim handling process: `employee where "Manager" in roles.roleName`. For automatically executed activities, like already mentioned "Check Insurance", performer query would come down to `AUTO`, so that the process management engine will know that this is an automatic activity.

attributes - collection of attributes that hold data specific to particular activity, and can be accessed from within other activities of the process. One attribute can be either a single object or a collection of

objects. The "Check Insurance" activity definition can for example hold a collection of policies found in the database, that could be used by the subsequent activities.

resources - specification of resources that this particular activity will require in order to be successfully carried out. This definition consists of a resource identifier and needed quantity. Demands like "20 computers" or "5 trucks" can be expressed within this element. "Send Letter" activity may, for example, require a computer workstation with printing device attached to it, together with some paper for printing.

In Fig. 13, a definition of "Phone Garage" activity is illustrated as it would be stored in SBA data store.

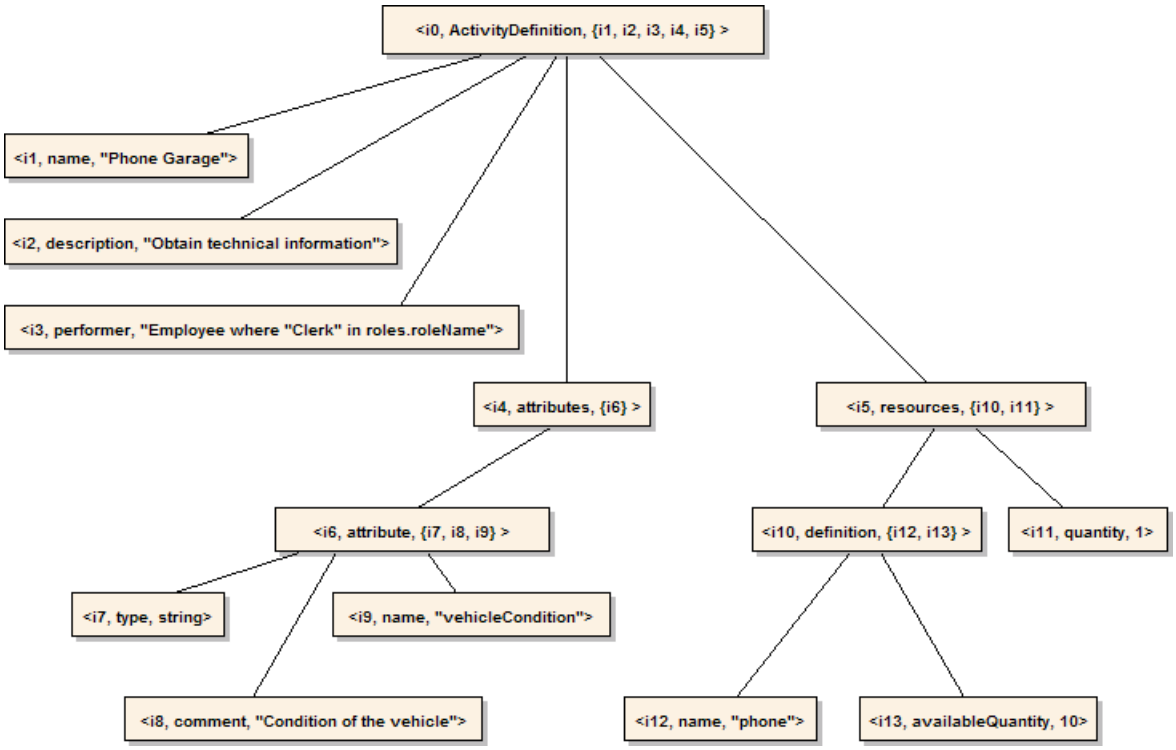

Figure 13: Example activity definition

Activity instance

When an activity definition gets instantiated, an activity instance comes to life. Those objects contain most of the process routing logic, enclosed within pre- and postconditions. Activity instances can also have references to other processes, nested inside. Other important aspects of those elements include references to following activities, i.e. activities that can be executed (but do not have to - depending on their preconditions) after the current one completes. Activity instances also refer to their immediately preceding activities, which can serve for checking their statuses or attached attributes.

name - name of the activity, meaningful and externally readable. Example names for an activity instance can be "Check Insurance", or "Register Claim".

description - description of the activity instance. Just like description field of the activity definition, it can contain detailed information about what should be done within this activity or what is its purpose. Unlike description of an activity definition, which spans all instances of particular activity, this field should contain instance-specific details.

definition - by containing this reference, we denote which activity definition should be instantiated. One definition can have many instances, and one particular instance may have exactly one definition. Activity instance named "CheckInsurance" can for example refer to "CheckInsuranceDefinition", as its definition.

preconditions - this query will be helpful in evaluating preconditions of the activity, i.e. conditions that must be satisfied in order to proceed with execution of this activity instance. This element can be just any SBQL query that returns a boolean value. Preconditions play the key role in determining routing between activities, and they can also act as high-level synchronization blocks. There can be many conditions connected with logical operators; there is also an option of signaling the workflow management system to wait until particular conditions are satisfied (eg. invoice from other company is received). Example preconditions specification can be as simple as `true` (activity will always be executed), or more complex. Let's consider our sample process. We want to express that "Pay" activity could only happen if the earlier made decision was positive. Thus, preconditions section of "Pay" activity should read: `(this.precedingActivities where name = "Decide").paymentDecision`, where `paymentDecision` would be an activity attribute that has been set to true or false during that activity's execution.

postconditions - this section is similar to the above described preconditions. The only difference is that the postconditions are evaluated after the activity instance has finished execution. Again, let's consider the "Decide" activity. By specifying postconditions, we want to express what is enough for particular activity to complete. In this situation, "Decide" activity would be finished if, naturally, payment decision had been made. For denoting this, we use already mentioned `paymentDecision` attribute and set its value to true or false, so that another activities could check it. Therefore, our postconditions section of the "Decide" activity would read: `this.paymentDecision = true or this.paymentDecision = false`. Null value in this place would mean lack of any decision, and this state we want to prevent.

processes - nested subprocesses that are to be executed within this activity instance. Processes can be infinitely nested (of course without considering resources, which are always finite), and every section with nested processes can contain any number of sibling subprocesses (processes that are on the same level of nesting). This approach supports hierarchical alignment of workflows and fosters reuse - already defined processes can be easily plugged into existing activities. In the claims handling process, good candidate for nested subprocess would be that related to executing payment, within "Pay" activity. Activities of that subprocess could involve invoking external banking system and making money transfer.

followingActivities - collection of activity instances that will have their preconditions evaluated, and (in case of positive result) will be executed in parallel by default, if only available resources suffice. This section can contain only references to activity instances, all the routing constructs and logic can be modeled by using pre- and postconditions (described earlier) and including them in each element

of the following activities' collection. For example, "Classify" activity instance will have three elements in the `followingActivities` collection: "PhoneGarage", "CheckInsuranceSimple", and "CheckInsuranceComplex". This does not mean that each of those three activities will be executed, but preconditions of all three will be evaluated, and relevant process route will be selected depending on the outcome.

precedingActivities - collection of references to activities that have already been completed before the current activity started execution. This element can be useful when an activity depends on previous activities' statuses or there is a need to access data contained in preceding activities. Example reference to previous activities' attributes has been shown in preconditions' description paragraph, where we wanted to access the `paymentDecision` attribute from within "Pay" activity instance.

performer - reference to the instance of `Performer` class, which represents resource already allocated as a result of `performer` query evaluation, specified in current activity's definition. "Decide" activity of our example business process would have concrete object of class `Performer` allocated, containing data about specific manager that is responsible for making this decision.

resources - collection of resource objects that are both required and allocated by the current activity instance. Allocated resources cannot be reallocated by another activity instance, unless they are deallocated first. This can happen upon activity completion or failure. Let us say, that the "Send Letter" activity from our sample process will consume one computer workstation with a printer. In this case, the resources section will include a computer with quantity 1, and a printer with the same quantity. After the activity has been completed, both of those resources will return to the pool of available resources, ready for another allocation.

attributes - data objects that are attached to current activity instance (activity-scoped variables). Attributes can be accessed from within this or any other activity instance, or any process that is being executed within the same workflow management system instance. Processes may have different routing paths, whose selection is determined according to this data (data-based routing). Also, pre- or post-conditions may depend on particular attribute's presence or value. Attributes support wide spectrum of values: from primitives, like strings, numbers, to more complex ones - whole documents, multimedia files, or aggregated data objects. As an example, we can imagine that activity named "Register Claim" would have a digitalized claim document attached to it as an attribute. Then, all the subsequent activities could access this document in order to retrieve needed information.

startDate - date and time when current activity instance began execution.

finishDate - date and time when current activity instance ended execution.

status - status of the activity instance. Can contain simple constants ("WAITING", "FINISHED", etc.) or can be dynamically evaluated at run time, as a result of more complex SBQL queries. The "Check Insurance" activity of our example process will have its status evaluated according to the presence of attributes that could potentially contain insurance policies data, so that the following activities could easily check that attribute and determine further processing.

instances - query that should return number of instances of this activity to execute. Example queries for this section can be `3`, `2+2*5`, or `count(customers where city = "Warsaw")`, hence any valid SBQL query that returns an integer. This can be useful when we want to carry out a particular task, but for different cases - our example "Send Letter" activity could be executed three times, if there were three different originators of the same insurance claim.

errors - collection of errors that have been reported during execution of this activity instance. After the process finishes, those errors are checked against specified exception handlers in order to run relevant compensation procedures. Concerning the example process, far-reaching consequences could result from an error during "Pay" activity, so all such situations need to be considered and relevant actions should be taken.

An example of activity instance is shown in Fig. 14. It illustrates how "Phone Garage" task could be stored in SBA data store. Note that in this diagram, not all elements of activity instance have been used. That is because some of them are not applicable for this particular task (i.e. work element, as it is not an automatic activity), or have not been used (i.e. subprocesses or errors).

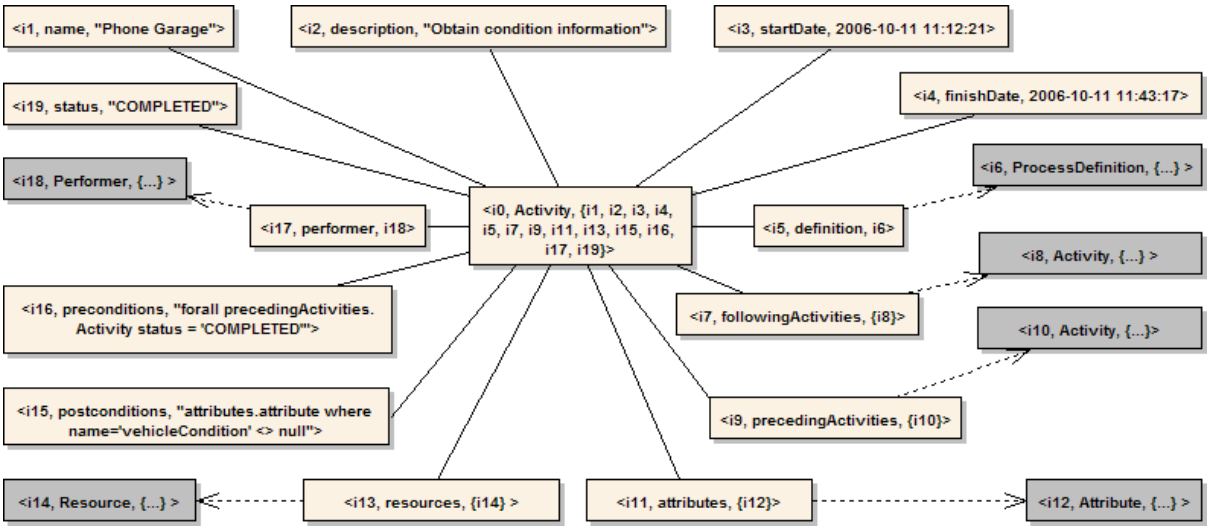

Figure 14: Example activity instance

Process definition

These objects contain activities that need to be carried out within all instances of the current process definition, together with compensation procedure triggered in exceptional situations, depending on the errors that occurred in each of this process' activities.

activities - set of activities that make up this process. By default, all activities specified within this section are to be executed in parallel. If this is not a desirable behavior, more sophisticated routing rules can be defined using each activity's preconditions. Each element of this collection can have `followingActivities` section, specifying which activities need to be attained afterwards.

attributes - all possible data objects that can be attached to instances of this process, should be defined here. Values of attributes specific to process instances should not be kept here, although the default values can be. A process-scoped attribute can be, in case of the insurance claims handling process, personal details of the claim originator. Therefore, any activity within this process could directly access this information.

resourceAllocation - one of several methods for allocating resources, according to which the system's resources management module would allocate tasks. Value of this element can be, for example, `"load_balance"`, `"random"`, or `"round_robin"`.

compensation - business logic related to handling errors that occurred during execution of a process. Compensation procedure can be simple, for example only report the errors and send email to appropriate people, or more complex - invoke external systems to roll back changes made within this process instance. Failure of an insurance claim to register, for example due to lack of important information on the claim form, should immediately be reported back to the originator. Process created to handle this specific claim should then be canceled. This would leave the system in the consistent state, and would prevent further errors during execution of this process instance.

Process definition object of the insurance claims handling process could be written and stored in the data store, as it is depicted in Fig. 15.

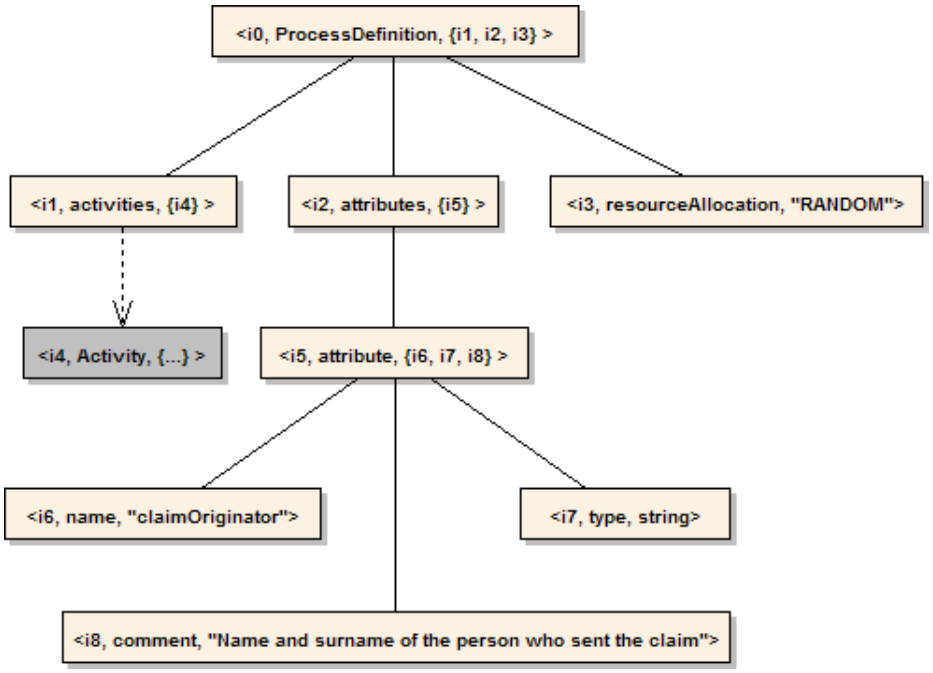

Figure 15: Example process definition

Process

Instance of the process is one of the simplest objects in this extension. It doesn't contain any business or routing logic itself, because all of it has already been defined in process definition and activity instance

objects.

name - externally readable and meaningful name of the process instance.

definition - reference to the definition of this process. One process instance can have exactly one definition, and one definition can be instantiated repeatedly. For example, the claim handling procedure, defined once, can be instantiated for each incoming insurance claim.

startDate - date and time when the process instance has begun execution.

endDate - date and time when the process instance has ended execution.

attributes - collection of process-scoped attributes, that comprise data objects related to process. These objects can be accessed from within any activity enclosed by this process instance, along with other process instances.

Structure of a sample process instance written to SBA data store, is illustrated in Fig. 16.

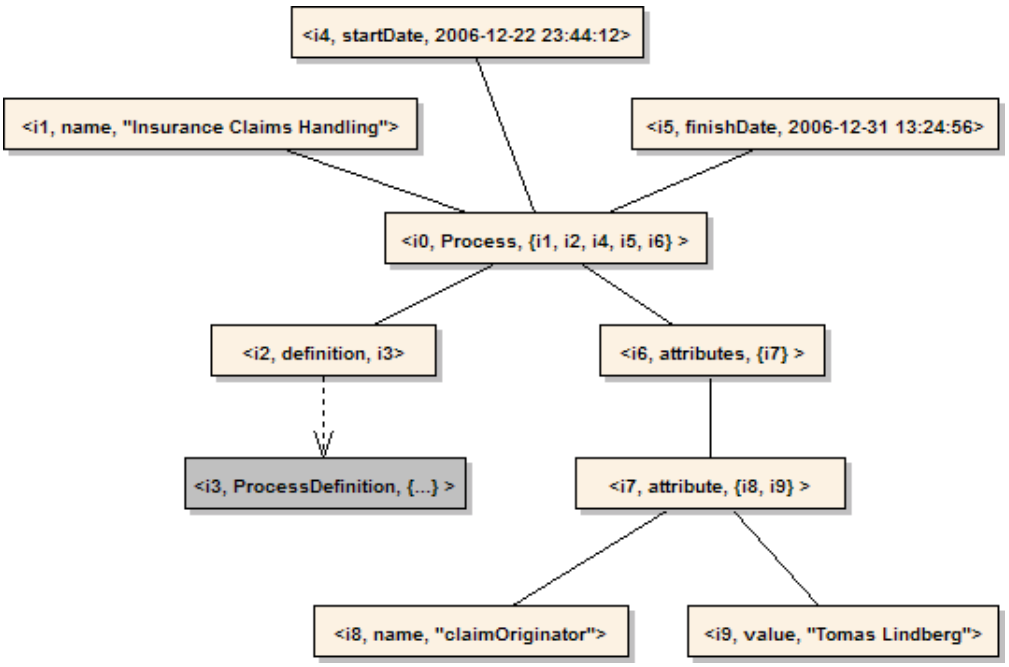

Figure 16: Example process instance

Attribute

Processes or activities in this SBA/SBQL extension may contain attributes. Those are data objects according to which routing paths may be chosen, but they can also serve as simple containers for data manipulated by subsequent activities or nested subprocesses.

type - this field denotes type of the attribute. It can be a primitive type, or a class name. Values of the attribute will be constrained by this field, i.e. the value field will permit only legal values according to the type. Example types include `integer`, `string`, `ClaimOriginator` or `InsuranceClaim`.

name - name of the attribute that will be used to manipulate its data. This field is externally readable and meaningful. Typical attribute names would include `initialDocument`, or `insuranceCase`.

comment - additional comment that would inform the reader about the nature of this attribute. May be much longer than attribute's name, but lengthy descriptions are discouraged in this place.

value - value of the attribute. Any object that conforms to the `type` field. Can be retrieved or set from within activities or processes.

Resource definition

These elements can be created and stored in order to keep information about availability of specific resources. Fig. 17 shows an example structure of phone resource availability definition that is used in the insurance claims handling process.

name - readable and meaningful name of the resource that is being defined. Example resource names: "truck", "computer", or "printer".

availableQuantity - quantity of specific resource that is currently available for allocation. By specifying this value we mean the global amount of resources that is available for use immediately during activity execution.

Resource

Objects of this type will specify how much of particular resource is needed to carry out an activity. Resource objects can be kept inside the activity instances, denoting amount and kind of resources allocated to the current task.

definition - reference to the definition of resource, where more contents pertaining availability, quantity, descriptions and other details are kept.

quantity - amount of items that represent this resource. This field is usually specified to denote the demand for particular resource. When a resource gets allocated to activity, value of this element is subtracted from the available resource's quantity, and it is added again after deallocation.

2.3 SBQL extensions for defining business processes

Supplementary to the SBA data store enhancements is the Stack Based Query Language extension, which proposes adding several new keywords for creating and manipulating business processes. It consists of special operations that will be performed on process definitions, process instances, activity definitions, and activity instances. Syntactic rules of the proposed extensions together with their semantics will be explained in subsequent sections, using example that has already been introduced in previous section.

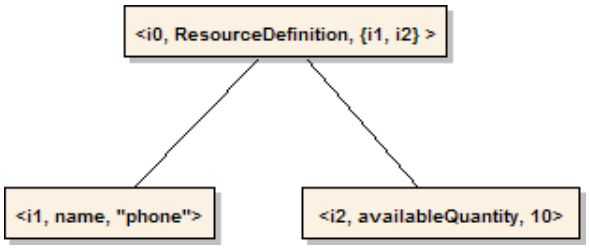

Figure 17: Example resource definition

Creating process definition

Designing processes, or creating their definitions is probably the most crucial part in the overall process of workflow-based information system design. Once the definitions are created, numerous process instances can be created according to them at runtime. SBQL extension proposed within this work assumes existence of two major elements needed to compose such definition: `activities` that will contain all the tasks enclosed by this process, and `attributes` for defining data containers that will hold process scoped variables.

Declarative exception handling mechanism also uses the `create process definition` statement for defining erroneous situations management rules, which can be specified with the `compensation` keyword. This procedural section of a process definition can be used to invoke simple actions, like reverting changes made to persistent store, or some more complex and human-involving operation, like sending second letter, after the first one turned out to contain erroneous or irrelevant information.

Example code below creates process definition for handling insurance claims.

```
create process definition {
    name "Handle Insurance Claim";
    allocation "RANDOM";
    activities {
        bag { registerClaim }
    }
    attributes {
        case_number : string;
        reg_number : string;
        vin_number : string;
        claim : &Document;
    }
} as handle_claims_def;
```

The above code will create a process definition with name "Handle Insurance Claim", one initial activity - `registerClaim`, and four attribute definitions for holding case-specific information, like claim number, vehicle registration number, VIN, and original claim document in digitalized form. All the instances of this process will have their activities randomly distributed among eligible performers, as it has been specified by the `allocation` keyword. From now on, developers will be able to instantiate and run processes according to this definition. It is important to note that prior to creating process definition all its activities need to be created. Another important fact is that although very useful, `compensation` keyword has been omitted in this example in favor of code clarity and comprehensibility.

Creating process instance

Once the definition has been created, one can now instantiate it. Each process instance is a single case, handled individually. Provided that we have executed the above code, we can create its single instance by executing the following SBQL command:

```
create process {
    name "Handle Insurance Claim no. 244";
    definition handle_claims_def;
} as handle_claim_244;
```

It will create a new instance in accordance to definition specified in the previous section. All that is required to achieve this, is providing an appropriate reference to the already existing definition, in this case `handle_claims_def`. Process instance named "Handle Insurance Claim no. 244" is now ready to be started and executed.

Creating activity definition

Activity definition is used analogously to the process definition - it specifies common elements to all activities representing this type, and according to it, new task instances will be created. Activity definition is slightly more complex than two previously discussed elements. Therefore, creating it is a bit more complicated, yet sill remains comprehensible and fairly easy to use. In order to create an activity definition, several elements must be specified.

Probably the most important part of any task definition, or even any process running within the control of workflow management engine, is the procedural section, which is defined by the `work` clause. Inside it, programmers should write SBQL code containing business logic, which could be some heavy data processing, calculations, or simply delegation to other procedures or external systems. However, this code should not contain any logic related to process routing, which is specified in other elements that will be described later in this chapter. The code enclosed within the `work` clause will be executed by the workflow engine upon task enactment. In the example claims handling process, the "Check Insurance" activity could contain procedure that searches through the insurance database and, if any insurance is found, puts result into appropriate activity-scoped attribute, making them available for later retrieval.

Another important statement, denoted by the `performer` keyword, is used to dynamically select performer of activities sharing this common definition. Leveraging SBQL query also in this section results in greatly increased flexibility, as the performer query can be virtually anything that evaluates to existing resource, or, in case of tasks carried out automatically, a special string - `AUTO`. In the sample process, "Check Insurance" task could be executed automatically, hence its `performer` section would only contain `AUTO`, whereas the "Phone Garage" activity would involve human action to be taken, so its `performer` section would contain the following query: `Employee where "Clerk" in roles.roleName`.

As it was stated earlier, the proposed extension also includes infinite nesting of processes. Activity definition has been chosen as the integration point between parent and child processes. In order to define such relation, one has to use the `subprocesses` section inside the `create activity definition` statement, and supply a collection of references to existing process instances. Those subprocesses can also be supplied in the form of an SBQL query that will be evaluated at runtime. Such degree of freedom and generality helps in achieving true dynamically changing process definitions, and enables designers to defer some decisions until runtime, which also boosts flexibility. The Insurance Claims Handling example, being a relatively simple process, does not leverage the subprocess nesting

feature. However, most of the tasks that it consists of, can be decomposed, thus comprising many fine-grained processes. A good candidate for such decomposition could be the "Pay" activity, consisting of several small subtasks, like "Fetch Client's Account Number", "Transfer Funds", and "Log Transaction Event".

Vital to proper workflow handling is appropriate resource management. Inside every activity definition, programmers may indicate demand for non-human resources by writing relevant SBQL query inside the `resources` section. Each instance, right before execution, will be given those required resources, as long as they suffice. Upon completion of such resource-involving task, allocated objects will be deallocated by the resource management module, and sent back to global resource pool. "Send Letter" activity, for example, requires a computer with printer and one envelope.

Example activity definition below illustrates usage of elements discussed in this section. This code creates definition of "Phone Garage" activity, of the insurance claims handling process.

```
create activity definition {
    name "Phone Garage";
    description "Phone the garage in order to obtain necessary "+
                "information about condition of the vehicle";
    performer {
        Employee where "Clerk" in roles.roleName;
    }
    attributes {
        vehicleCondition : string;
    }
    resources {
        Resource where category = "Phones";
    }
} as phone_garage_def;
```

What has been created above, is the activity definition with performer chosen by role, and one attribute of string type. All instances of this definition will try to allocate one item of category "Phones" as resources necessary to carry out the task. Noteworthy is the fact that in the above code, sections like `work` and `subprocesses` have been left out. That is because they were not required for this particular definition - task is not meant to be automatically executed, so `work` section is useless. There are also no subprocesses expected in activities representing this definition, hence no `subprocesses` section.

Creating activity instance

Creating an activity instance according to its definition will provide a task that is ready to be placed in processes and executed by the workflow management engine. In addition to referencing the definition, activity instance also includes elements required to properly handle process routing, like discussed earlier pre- and postconditions, together with specification of activities that ran before this particular task and which should be enacted after it completes.

The `preconditions` and `postconditions` sections are evaluated before the core task functionality is run, and can be used as high-level synchronization mechanisms. Both of those sections require programmers to write SBQL queries that return either true or false. In the latter case, when preconditions are evaluated, such activity will not be performed. However, programmers may specify, by using `waitfor` clause, that the workflow management engine should wait for the preconditions to

be satisfied. Similarly, process management system may also wait for the postconditions to be met. This can be useful when further processing is dependent on presence of specific data or its value.

Activity instances may point to other activities to denote that they need to be carried afterwards. To achieve this, designer or programmer should provide collections of references to tasks inside the `following_activities` clause. It can consist of either several, one, or no elements at all. The latter case is used to indicate implicit end of the process, as no further activities are eligible for execution.

Process routing often depends on previous activities' statuses or attributes, hence there has been identified need for specifying immediately preceding tasks' identifiers in `preceding_activities` section. The only difference between `preceding_activities` and `following_activities` sections is that the former requires specification of activity identifiers only, whereas the latter needs references to regular activity objects. This has been introduced to avoid cyclic references between task instances.

Let us now consider an example activity instance specification, based on the Insurance Claims Handling process.

```
create activity {
    name "Classify Claim";
    description "Classify claim as either simple or complex";
    definition classify_def;
    preconditions {
        waitfor {
            "COMPLETE" in (Activity where name = "Register Claim").status;
        }
    }
    postconditions {
        waitfor {
            (attribute where name = "complexity").value <> null;
        }
    }
    preceding_activities {
        bag { register_claim }
    }
    following_activities {
        bag { phone_garage, check_insurance }
    }
} as classify_claim;
```

This code has created a new instance of the "Classify Claim" activity, according to definition passed in the `definition` clause. The task will execute only if the previous activity has completed, i.e. when its status is equal to `COMPLETE`. Then, procedural part will be invoked and at the end, postconditions will be evaluated - in this case, the workflow management system will check whether the attribute named `complexity` has been set. This `create activity` declaration also specifies that immediately preceding activity was `register_claim`, and next to the "Classify Claim" task execution, two activities should get their preconditions evaluated, and, conditionally, carried out.

Manipulating processes

Once the process has been composed, with all its activities defined and instantiated, it not needs to be started. Therefore, the proposed SBQL extension introduces the `launch` keyword which informs the workflow management engine that particular process instance needs to be started. For the example

Claims Handling Process, launch procedure would look like this:

```
launch handle_claim01;
```

The `handle_claim01` passed as a parameter is a reference to an existing, but not already started or completed, process instance. Upon reception of process launch instruction, the workflow management system will take actions to run procedural parts of activities, select execution paths, evaluate conditions, allocate resources and handle exceptional situations. Sequence and structure of those actions is illustrated in the activity diagram shown in Fig. 18.

Running process instances sometimes need to be canceled, for example to avoid or resolve deadlocks, or when something goes wrong and there is no sense in further execution. This can be achieved by issuing the following command:

```
cancel handle_claim01;
```

Now, process management system will stop performing `handle_claim01` instance and deallocate all resources that it consumed.

Both the `launch` and `cancel` reserved words are allowed to be used within activities' procedural sections, hence making process execution even more flexible and powerful.

Workflow processes that are kept within the data store, as well as all other entities related to them, like activities, resources, etc., can be queried externally during the entire process lifecycle, i.e. even after process completes or before it starts its execution. Those additional manipulation procedures are seamlessly integrated with existing SBQL queries or constructs, making them easy to learn and use by the developers. Access to this functionality can be helpful in managing running processes, resolving problems, like deadlocks, or analyzing possible resource shortages.

Not only administrators can leverage querying or manipulating functions. Application developers who use this workflow system as an underlying process engine would, for example, want to fetch all tasks assigned to given user (to render his personal work list in the user interface), get all process instances that are currently being carried out by particular group of performers, or use it as an interface to add/remove resources.

In order to fetch all activity instances that are currently waiting for user actions, one has to issue the following command:

```
Activity where status = "NOTIFIED";
```

On the other hand, when one wants to display the full list of processes sharing the same process definition, no matter whether they are currently being executed, finished or not even started, below query could be helpful:

```
Process where "Insurance Claims Handling" in definition.name;
```

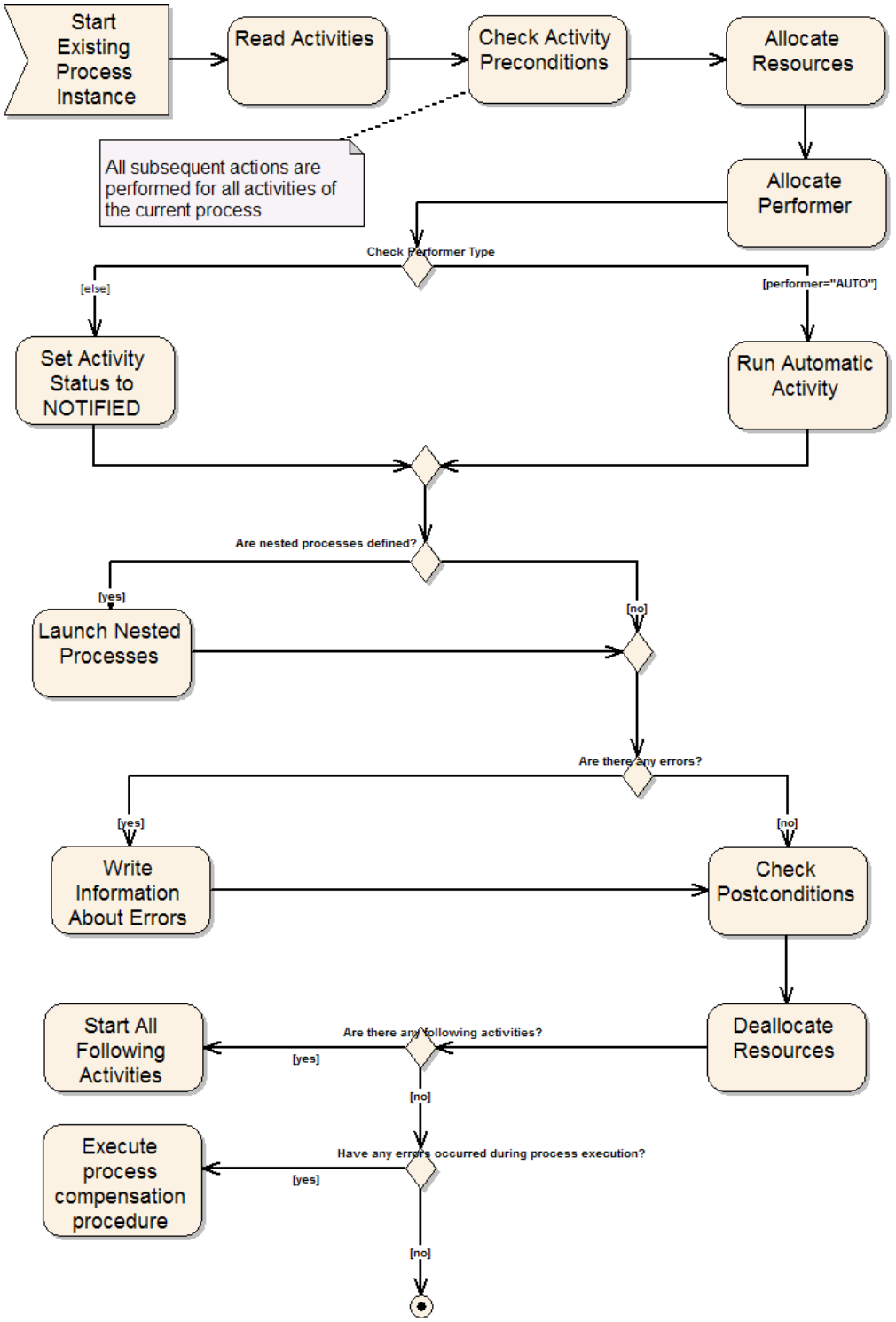

Figure 18: Process Execution

Although those two examples can be useful, real life queries or process manipulation instructions can get much more complex. For example, the below query can be help analyze which activities consume significant amounts of resources.

```
Activity where forany (resources.resource)
    quantity > definition.availableQuantity / 2;
```

Another example includes modifying the existing, and possibly running process. During an ongoing credit process at the bank, the management board could pass a bill saying that all credit applications need the supervisor's acceptance and, additionally, require sending a letter to the applicant. Traditionally, the procedure could begin for all new credit processes, started after the bill has been enacted. But there is a better possibility - the process designers can change the currently executing instances so that they reflect the legal status. The SBQL query shown below can be used to illustrate this action.

```
(Activity where name = "Accept Credit").followingActivities :=
bag { supervisor_acceptance, send_approval_letter };
```

As it can be observed from the examples above, the SBA entities introduced within this work can be handled exactly the same way, as all other objects present in the store. However, attention has to be paid to the fact that words like `Activity`, `Process`, and so forth, are reserved, and thus cannot be used as identifiers, or against their syntax. The complete list of reserved words that has been introduced to SBQL within this thesis, is placed in Appendix A.

The complete process

The example Insurance Claims Handling process that has been introduced in the beginning of this section, can be defined and instantiated by invoking the SBQL code presented below. Result of this code's execution, as it would be kept in the data store, has been placed in the Appendix A, in the form of Stack-Based Approach notation.

```
create activity definition {
    name "Register Claim";
    performer {
        Employee where "Clerk" in roles.roleName;
    }
} as registerClaimDef;

create activity definition {
    name "Classify";
    performer {
        Employee where "Manager" in roles.roleName;
    }
} as classifyClaimDef;

create activity definition {
    name "Phone Garage";
```

```
    performer {
        Employee where "Clerk" in roles.roleName;
    }
    resources {
        Resource where category = "Phones";
    }
} as phoneGarageDef;

create activity definition {
    name "Check Insurance";
    performer {
        "AUTO";
    }
    work {
        // SBQL code
    }
    attributes {
        policiesFound : &Policy[0..*];
    }
} as checkInsuranceDef;

create activity definition {
    name "Check History";
    performer {
        "AUTO";
    }
    work {
        // SBQL code
    }
    attributes {
        vehicleHistory : string;
    }
} as checkHistoryDef;

create activity definition {
    name "Decide";
    performer {
        Employee where "Manager" in roles.roleName;
    }
} as decideDef;

create activity definition {
    name "Pay";
    performer {
        Employee where "Clerk" in roles.roleName;
    }
} as payDef;

create activity definition {
    name "Send Letter";
    performer {
        Employee where "Secretary" in roles.roleName;
    }
    resources {
        Resource where category = "Paper";
        Resource where category = "Pen";
    }
} as sendLetterDef;
```

```
create activity {
    name "Register Claim";
    definition registerClaimDef;
    following_activities { bag { classifyClaim } }
} as registerClaim;

create activity {
    name "Classify Claim";
    definition classifyClaimDef;
    preconditions { registerClaim.status = "COMPLETED" }
    preceding_activities { bag { registerClaim } }
    following_activities { bag { phoneGarage, checkInsurance }
} as classifyClaim;

create activity {
    name "Phone Garage";
    definition phoneGarageDef;
    preconditions { classifyClaim.status = "COMPLETED" }
    preceding_activities { bag { checkHistory, classifyClaim } }
    following_activities { bag { decide } }
} as phoneGarage;

create activity {
    name "Check Insurance";
    definition checkInsuranceDef;
    preconditions { classifyClaim.status = "COMPLETED" }
    preceding_activities { bag { classifyClaim } }
    following_activities { bag { checkHistory, decide } }
} as checkInsurance;

create activity {
    name "Check History";
    definition checkHistoryDef;
    preconditions { checkInsurance.status = "COMPLETED"
                    and classification.value = "COMPLEX" }
    preceding_activities { bag { checkInsurance } }
    following_activities { bag { phoneGarage } }
} as checkHistory;

create activity {
    name "Decide";
    definition decideDef;
    preconditions { phoneGarage.status = "COMPLETED"
                    and checkInsurance.status = "COMPLETED" }
    preceding_activities { bag { phoneGarage, checkInsurance } }
    following_activities { bag { pay, sendLetter } }
} as decide;

create activity {
    name "Pay";
    definition payDef;
    preconditions { decide.status = "COMPLETED"
                    and decision.value = "PAY" }
    preceding_activities { bag { decide } }
} as pay;

create activity {
    name "Send Letter";
    definition sendLetterDef;
```

```
    preconditions { decide.status = "COMPLETED" }
    preceding_activities { bag { decide} }
} as sendLetter;

create process definition {
    name "Insurance Claims Handling";
    allocation "RANDOM";
    activities {
        bag { registerClaim }
    }
    attributes {
    classification : string;
    decision : boolean;
    }
} as handleClaimDef;

create process {
    name "Handle Insurance Claim no. 244";
    definition handleClaimDef;
} as handleClaim244;
```

3 Supported structures and workflow patterns

There has been a lot of research in the field of workflows. Despite standardization efforts, there are also numerous vendor-dependent or academically developed management systems and notations. From this differentiation emerged the need to describe common elements of most workflow notations and solutions. The notion of **workflow pattern**, derived from design pattern as we know it from object-oriented design, has been introduced in the work [Aalst et al.]. As the name suggests, workflow pattern is a common solution to recurring problem. These patterns are usually divided into three major groups. Control flow patterns convey solutions to problems pertaining process routing logic, resource patterns deal with performer and resources allocation, while data patterns correspond to various operations that are performed on data attached to processes or activities.

The workflow extension for SBA/SBQL that has been described in the previous chapter, having its notation and process enactment logic flexible and generic, naturally supports numerous workflow patterns. As it has been stated earlier, additionally to the SBA/SBQL extensions, a Java-based implementation of those extensions has been developed within this work. It uses XML markup controlled by XML Schema definitions for defining and storing processes, as well as activities with their definitions. In the following sections of this chapter, we will discuss support for workflow patterns in the proposed SBQL extension, as well as in the XML notation developed for the needs of the Java-based prototype. Detailed overview and description of the implementation itself will be described in the next two chapters.

3.1 Control flow patterns

Control flow patterns are probably the most widely-known among all workflow patterns. These constructs are usually very simple, hence commonly understood. They have been applied in workflows long before anybody had called them patterns. However, some of the control flow patterns are fairly complicated and harder to comprehend, thus will be discussed more thoroughly.

Basic flow constructs The simplest alignment of activities in a process is the **sequence** pattern. As the name indicates, activities are executed sequentially, one after another. In the developed prototype, sequence can be easily achieved using preconditions - each activity will reference another within the following activities section, and each one, will have to check precondition that the previous activity has completed. SBQL and XML fragments listed below illustrate sequence consisting of two activities.

```
create activity {
    name "a2";
    ...
    preconditions {
        waitfor {
            (Activity where name = "a1").status = "COMPLETED";
        }
    }
} as a2;

create activity {
    name "a1";
    ...
    following_activities {
        bag { a2 }
    }
} as a1;
```

```
<activity>
  <id>a1</id>
  ...
  <following-activities>
    <activity>
      <id>a2</id>
      ...
      <preconditions>
        <condition wait="true">
          <status>
            <activity>a1</activity>
            <value>COMPLETED</value>
          </status>
        </condition>
      </preconditions>
    </activity>
  </following-activities>
</activity>
```

Another elementary pattern reflects split of control into two or more threads of execution. Consecutive activities are accomplished in parallel, hence the pattern is called **parallel split**. The created workflow solution supports this pattern natively, without the need to explicitly specify the split. It can be modeled by just placing several activities in the following activities section. The workflow management system will by default carry them out in parallel. The snippets shown below describe process fragment with activities B and C being executed in parallel, after execution of A. General, static structure common to all split/choice patterns is illustrated in Fig. 19.

```
create activity {
    ...
    following_activities {
        bag { B, C }
    }
} as A;
```

```
<activity>
  <id>A</id>
  ...
  <following-activities>
    <activity>
      <id>B</id>
      ...
    </activity>
    <activity>
      <id>C</id>
      ...
    </activity>
  </following-activities>
</activity>
```

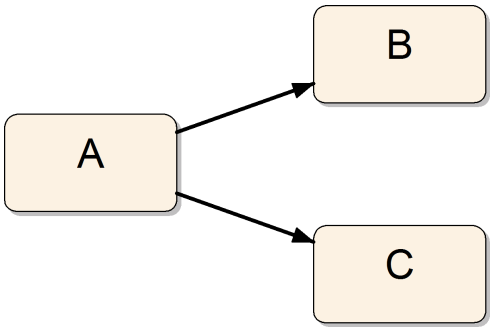

Figure 19: Split/choice patterns

Second construct, similar to the discussed above, is the **exclusive choice**. Unlike parallel split, this one permits only one of subsequent activities to be executed, depending on a decision made beforehand. This pattern can be easily achieved by using previous XML listing and adding relevant preconditions to activities A and B. For example, the choice may depend on process-scoped attribute named `decision` and set during execution of activity A. Then, precondition of B should read:

```
create activity {
    name "B";
    ....
    preconditions {
       "B" in (attributes.attribute where name = "decision");
    }
} as act1;
```

And, in the XML notation:

```
<preconditions>
  <condition>
    <attribute>
      <name>decision</name>
      <relation>EQ</relation>    <!-- equals -->
      <value>B</value>
    </attribute>
  </condition>
</preconditions>
```

Precondition of C would be similar, but it should check for different value inside the `attribute` tag: `<value>C</value>`.

The last pattern from the split/choice group discussed here is the **multiple choice**. It can be described as a general case of previous two patterns. Both kinds of behavior are permitted here: parallel split, as well as exclusive choice. Just like in those patterns, multiple choice can be modeled by appropriate preconditions, but this time conditions for both activities B and C can be satisfied at the same time.

Join/merge patterns Processes that have their execution paths parallelized, sometimes need to merge those split branches back into one. This can be done in two general ways: the parallel process routes are merged with synchronization, i.e. "thread" that has completed his work waits for the others, or, simply, without any synchronization. Four patterns, representing these two approaches, will be described next. General structure of those patterns is shown in Fig. 20.

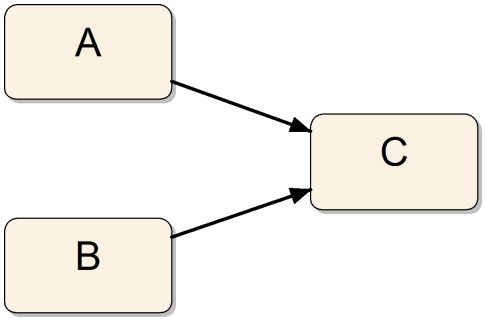

Figure 20: Join/merge patterns

The most straightforward merge pattern is called, naturally enough, **Simple Merge**. It is a point in the process where two or more branches converge into one. Let us consider this pattern using Fig. 20. Upon each execution of activities A or B, activity C will also get enacted, regardless of which preceding task has been completed first or whether they both finished their work at the same time. Important to mention here is the additional requirement of this pattern: regardless of the converging branches number, next activity will be executed only once (exclusive or). The developed workflow prototype notation can easily express Simple Merge. Given activities A, B, and C, the first two should both have activity C placed in their following activities section. Additionally, activity C should contain an exclusive or precondition, ensuring that only one of the preceding activities has been enacted. SBQL and XML notation examples for the Simple Merge pattern will look like this:

```
create activity {
    name "A";
    ...
    following_activities {
        bag { C }
    }
} as A;

create activity {
    name "B";
    ...
    following_activities {
        bag { C }
    }
} as B;

create activity {
    name "C";
    ...
    preceding_activities {
        bag { A, B }
    }
```

```
    preconditions {
       (((Activity where name = "A").status = "COMPLETED") or
       ((Activity where name = "B").status = "COMPLETED")) and
       (((Activity where name = "A").status <> "COMPLETED") or
       ((Activity where name = "B").status <> "COMPLETED"))
    }
} as C;
```

```
<activity>
  <id>A</id>
  ...
  <following-activities>
    <activity>
      <id>C</id>
      <preconditions>
        <condition>
          <xor>
            <condition>
              <status>
                <activity>A</activity>
                <value>COMPLETED</value>
              </status>
            </condition>
            <condition>
              <status>
                <activity>B</activity>
                <value>COMPLETED</value>
              </status>
            </condition>
          </xor>
        </condition>
      </preconditions>
      ...
    </activity>
  </following-activities>
</activity>
<activity>
  <id>B</id>
  ...
  <following-activities>
    <activity>
      <id>C</id>
      ...
    </activity>
  </following-activities>
</activity>
```

The **Synchronization** workflow pattern is a bit different from what we just discussed. Here, the two converging process paths are joined into one, thus synchronization between the two activities executed in parallel is achieved. Considering above example, in this scenario, activity A or B (depending on which one will be completed first) will have to wait until the other one finishes execution. Only then activity C will be activated, and will run only once. This behavior can be modeled similarly as the previous pattern example, but activity C should also contain additional preconditions:

- activity C should wait until both activities A and B complete execution
- status of activity C should be equal to `NONE`

The latter condition is essential for true synchronization to work. It ensures that activity C will not be carried out several times (for each converging branch), as the status constant `NONE` denotes that this particular activity instance has not yet been executed. Provided that any of branches that are being merged, will start this activity, its status will immediately be changed to `RUNNING`, thus no other "thread" will be able to perform this task. Therefore, preconditions section of activity C, in SBQL extension, would read:

```
preconditions {
    waitfor {
        ((Activity where name = "A").status = "COMPLETED") and
        ((Activity where name = "B").status = "COMPLETED");
    }
    ((Activity where name = "C").status = "NONE");
}
```

Whereas the same functionality written in XML notation would look like the following:

```
<preconditions>
  <condition wait="true">
    <status>
      <activity>A</activity>
      <value>COMPLETED</value>
    </status>
  </condition>
  <condition wait="true">
    <status>
      <activity>B</activity>
      <value>COMPLETED</value>
    </status>
  </condition>
  <condition wait="false">
    <status>
      <activity>C</activity>
      <value>NONE</value>
    </status>
  </condition>
</preconditions>
```

Sometimes there is a need to synchronize two process branches, but only if they both are activated (both started, but none of them has finished). In the opposite case, simple merge should be done, without any synchronization. This pattern is a combination of the two previously discussed - Synchronization and Simple Merge. **Synchronizing Merge**, because that is how it has been called, is a bit more complicated, but still can be represented in the notation developed for the needs of prototype. Preconditions specification is, just like in preceding cases, key to solving this problem. However, this time it cannot be depicted using pure XML. This reveals another feature of this workflow system, namely, conditions specified in the form of Java classes. Such class has the access to activity-scoped or process-scoped

data, and basically everything that can be accessed from within the XML process specification. At run time, condition classes are instantiated and appropriate methods are invoked to check whether those conditions have been satisfied or not. To apply Synchronizing Merge pattern, we need to construct the following precondition:

- if activity A is running, wait for it to complete
- else, if activity B is running, wait for it to complete
- in any other case, precondition outcome is true, hence it is satisfied

The last pattern that will be discussed from the merge patterns family, is the **Multiple Merge**. This one is very simple, as it represents general behavior of Simple Merge, a workflow pattern described above. Here, multiple branches of execution are joined, and any subsequent activities will be carried out as many times, as many branches the process had before merger. This pattern is naturally supported by the developed workflow notation and does not need any additional constructs. It is enough to define three activities and to place reference to one of them in two others' following activities section. Code snippets shown below illustrate this scenario.

```
create activity {
    name "A";
    following_activities {
        bag { C }
    }
    ...
} as A;

create activity {
    name "B";
    following_activities {
        bag { C }
    }
    ...
} as B;

create activity {
    name "C";
    ...
} as C;
```

```
<activity>
  <id>A</id>
  ...
  <following-activities>
    <activity>
      <id>C</id>
      ...
    </activity>
  </following-activities>
</activity>
<activity>
  <id>B</id>
```

```
  ...
  <following-activities>
    <activity>
      <id>C</id>
      ...
    </activity>
  </following-activities>
</activity>
```

N out of M Join and Discriminator Real life processes are rarely simple, and hence, they often have many parallel branches that need to converge into one at some point of execution. Activities that are next to this join point, need to be run only once, so only the first branch that finishes its previous work is taken into consideration, and any subsequent branches are ignored. A bit more complicated scenario may require completion of some specific number of preceding activities, in order to go on with execution of another activity. General behavior that adheres to those requirements can be provided by the **N out of M Join** pattern. In this construct, a specific number of parallel branches is required to complete execution before the next one will be enacted. Fig. 21 shows this described situation - tasks named from A to D converge into one process path, so that activity E will be run once only.

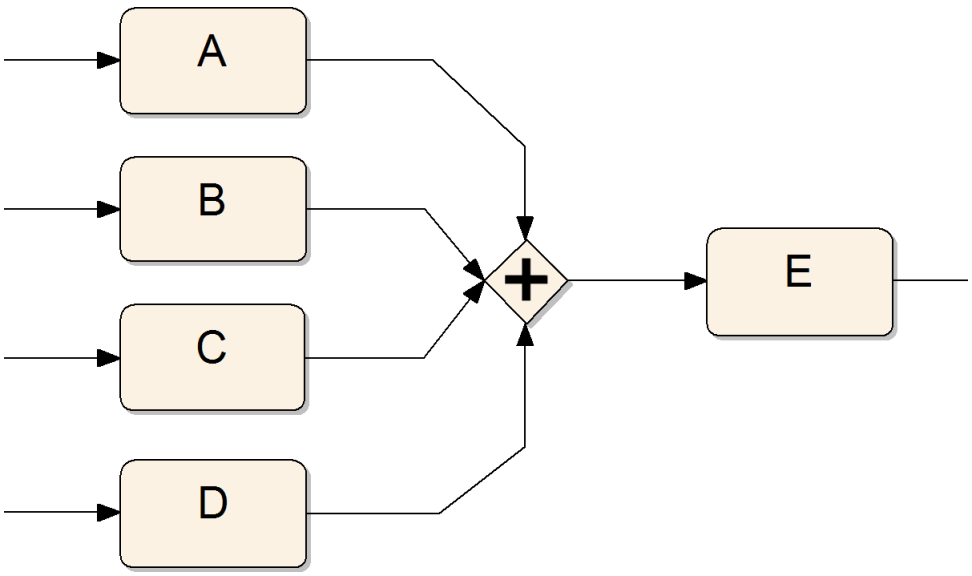

Figure 21: N out of M Join pattern

More specialized form of this workflow pattern is called **Discriminator**. Here, condition upon which next activity's execution depends, requires only one preceding activity to complete. Main difference between those two versions of a pattern lies in synchronization: in the first, general case, it is accomplished partially, whereas in the latter scenario, no synchronization is performed. The proposed Stack-Based Query Language extension, as well as XML notation developed for the prototype implementation supports these patterns explicitly, by enabling the process designer to specify exact number of activities that need to be finished for the next one to start execution. This condition should be placed in the preconditions section of the following activity, as defined below.

```
preconditions {
    count(precedingActivities) >= 3;
}
```

```
<preconditions>
  <condition>
    <completed-preceding>3</completed-preceding>
  </condition>
</preconditions>
```

This example condition will be satisfied only if three of the immediately preceding activities would have been completed, i.e. would have their status equal to `COMPLETED`. In order to achieve the discriminator pattern's behavior, condition value (or content of `<completed-preceding>` tag) needs to be changed to `1`.

Arbitrary Cycles Certain tasks or series of tasks require repeated execution, as long as relevant conditions are satisfied. This concept resembles notion of loops, known from the area of computer programming languages. It has been conveyed to the business process management field under the form of arbitrary cycles workflow pattern. Basic structure of this construct is presented in Fig. 22.

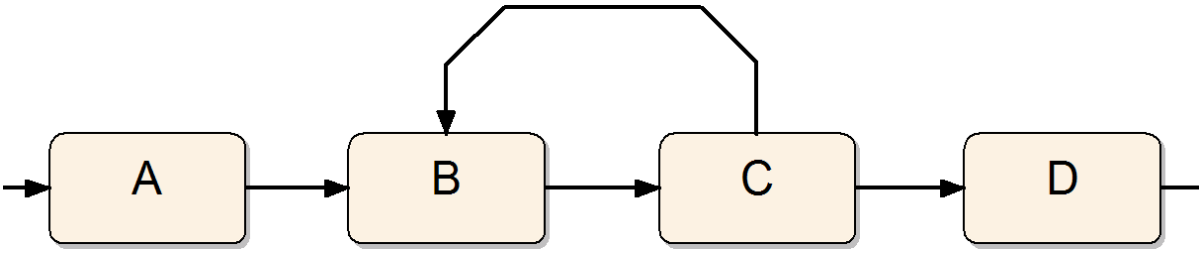

Figure 22: Arbitrary Cycles pattern

This pattern can be easily applied in the proposed extension, by using preconditions of consecutive activities. Let us consider alignment of tasks as in Fig. 22. We want to run activities B and C as long as C's attribute named `loop` is greater than `0`, assuming that each iteration of this loop decreases that variable by one. Therefore, preconditions section of activity B will contain the following condition:

```
preconditions {
    ((Activity where name = "C")
    .attributes.attribute where name = "loop").value > 0;
}
```

```
<condition>
  <attribute>
    <activity>C</activity>
```

```
    <name>loop</name>
    <value>0</value>
    <relation>GT</relation>    <!-- greater than -->
  </attribute>
</condition>
```

And, additionally, D will have to satisfy the following precondition in order to run:

```
preconditions {
    ((Activity where name = "C")
    .attributes.attribute where name = "loop").value <= 0;
}
```

```
<condition>
  <attribute>
    <activity>C</activity>
    <name>loop</name>
    <value>0</value>
    <relation>LE</relation>    <!-- less or equal -->
  </attribute>
</condition>
```

This last condition is necessary, otherwise task D would get carried out with every iteration of this loop.

Deferred Choice Defining the whole workflow process and all its details at design time severely impairs flexibility, as real-life processes tend to change rapidly, responding to events received from the environment, or even due to runtime calculations done within the process itself. That is why we need a mechanism for dynamic selection of process paths at execution time. The deferred choice pattern addresses those problems, enabling data-based routing of workflows. One, or possibly a few of process paths can be chosen according to, for example, presence of data, or its certain value. This mechanism is directly supported by the workflow system developed within this work. It enables not only dynamic routing based on existing, pre-set attributes, but also according to variables that can be set from within another activity. Let us consider a simple example, where one of two subsequent activities needs to be selected, basing on preceding activity's outcome. Thus, we need to define three activities. First one will do some calculations, upon which the choice will be done. Second and third activity will follow the first, and only one of them will execute, depending on the first task's attribute named `choice`. To model this scenario, we will use the following code:

```
create activity {
    name "A";
    ...
    following_activities {
        bag { B, C }
    }
} as A;
```

```
create activity {
    name "B";
    ...
    preconditions {
        ((Activity where name = "A").attributes.attribute where
        name = "choice").value = true;
    }
} as B;

create activity {
    name "C";
    ...
    preconditions {
        ((Activity where name = "A").attributes.attribute where
        name = "choice").value = false;
    }
} as C;
```

Or, the following equivalent XML markup:

```
<activity>
  <id>A</id>
  ...
  <following-activities>
    <activity>
      <id>B</id>
      ...
      <preconditions>
        <condition>
          <attribute>
            <activity>A</activity>
            <name>choice</name>
            <value>true</value>
            <relation>EQ</relation>    <!-- equal -->
          </attribute>
        </condition>
      </preconditions>
    </activity>
    <activity>
      <id>C</id>
      ...
      <preconditions>
        <condition>
          <attribute>
            <activity>A</activity>
            <name>choice</name>
            <value>false</value>
            <relation>EQ</relation>    <!-- equal -->
          </attribute>
        </condition>
      </preconditions>
    </activity>
  </following-activities>
</activity>
```

In the above example process fragment, activity A is responsible for making appropriate calculations and

setting its attribute `choice` (activity-scoped attribute). Next, the two following activities B and C will have their preconditions evaluated. As it is shown above, conditions in both of those activities depend on previously set attribute, and hence, only one task will be executed (B for `true`, C for `false`).

Implicit Termination Business process execution can be interrupted as a result of critical error, or their instances can be canceled at runtime due to specific circumstances. However, processes should also cease to run upon completion of all contained activities, when no other task can be enacted, and when the process is currently not in deadlock. This behavior is called implicit termination, and can be modeled using SBQL, as well as the prototype XML notation without much effort. The process manager module, responsible for launching processes and activities, will simply take no further action when there are no activities to execute left.

```
create activity {
    name "A";
    ...
    following_activities {}
} as A;

create process definition {
    activities {
        bag { A }
    }
} as processDef1;
```

```
<process>
  <id>process1</id>
  ...
  <definition>
    <activities>
      <activity>
        <id>A</id>
        ...
        <following-activities>
        </following-activities>
        ...
      </activity>
    </activities>
  </definition>
</process>
```

In this example, the following activities section is empty, denoting that there are no more activities defined within the process. Thus, process management engine will terminate this process implicitly.

Multiple Instances Business requirements often demand certain tasks to be performed several times, depending on data, external events or within some specified time period. Those work items may differ between each other (eg. processing each element from the list in separate activity) or may be required to act exactly the same way. Hence, there has been identified a group of workflow patterns that share

common name - multiple instances. Constructs from this category represent two main behaviors: multiple instances with a priori knowledge (i.e. predefined at design time or at runtime), or without a priori knowledge (calculated during process execution or as a result of external events) about number of instances. Both of those workflow pattern classes are covered by the developed prototype notation, and supported by the process manager module. When designer knows how many times specific activity is required to run, she may denote it as follows:

```
create activity {
    name "A";
    instances { 3 }
    ...
} as A;
```

Or, using the prototype's XML notation:

```
<activity>
  <id>A</id>
  ...
  <instances>
    <number>3</number>
  </instances>
</activity>
```

This means that activity A will be executed exactly 3 times, and this number cannot change at runtime.

Another case is when process designer does not know the exact number of instances, but knows how it can be calculated at runtime.

```
create activity {
    name "A";
    ...
    instances {
        (attributes.attribute where name = "inst").value;
    }
} as A;
```

```
<activity>
  <id>A</id>
  ...
  <instances>
    <attribute>inst</attribute>
  </instances>
</activty>
```

The above example shows a sample application of the pattern, without explicitly specifying number of instances. Attribute `inst` will convey required information, and task A will be instantiated many times according to it. This is a more flexible approach, as any preceding or external activity can change value of the `inst` variable, thus specifying number of instances to execute.

3.2 Resource patterns

Second, but equally important group of workflow patterns is related to dealing with resources. This is a vast area of workflow management and spans assignment of tasks to performers, as well as non-human resource allocation. Some patterns of this group, analogously to control flow patterns, have been applied since the beginning of business process management, without explicitly using their names or thinking as of patterns.

Direct Allocation The simplest, most straightforward and inflexible approach to task distribution is to assign activities to performers precisely specified at design time. Choice made that way cannot be undone, and the work item cannot be delegated to any other resource at process execution time. However, sometimes the need arises to constrain certain activities' performers to particular identity, rather than group or role. The direct allocation pattern enables this behavior, and it is naturally supported by the developed XML notation, as well as SBQL extension. Let us consider the following example:

```
create activity definition {
    performer {
        Employee where name = "JohnDoe01";
    }
    ...
} as ADef;
```

And the XML markup, which does the same:

```
<activity>
  <id>A</id>
  <definition>
    <performer>
      <direct>JohnDoe01</direct>
    </performer>
    ...
  </definition>
  ...
</activity>
```

By using `direct` performer type, it is specified that no other identity, but `JohnDoe01`, can carry out this task.

Role-Based Allocation Next workflow pattern slightly enhances flexibility, yet still building on the concept of previously discussed one. This time, designer has to specify a group name, which members will be capable of performing the task. At run time, the resource management module is responsible for selecting eligible users of specified role, and assigning this activity to one of them. Role-based allocation, because that is how this construct is called, is also supported in the developed workflow system. It has even been implemented in somewhat extended form, as it allows to denote several roles, each of which containing performers capable of accomplishing the activity. This concept is similar to the **authorization** workflow pattern, where multiple resources can be specified as capable of performing a task. Example of role-based allocation will look like the following.

```
create activity definition {
    performer {
        Employee where roles.roleName in
        ("Clerks", "Managers", "Analysts");
    }
} as ADef;
```

```
<activity>
  <id>A</id>
  <definition>
    <performer>
      <roles>
        <role>Clerks</role>
        <role>Managers</role>
        <role>Analysts</role>
      </roles>
    </performer>
    ...
  </definition>
  ...
</activity>
```

As we can see, task A can be carried out by any resource that belongs to role `Clerks`, `Managers`, or `Analysts`. Specific identity will be chosen at run time, depending on current resource allocation algorithm that will be described in the next chapter.

Deferred Allocation In previous section, we have discussed the Deferred Choice pattern, from the control-flow patterns group. Deferred Allocation is nothing more, but a construct that resulted from porting Deferred Choice concept to the area of resource management. Here, choosing performer for specific task is delayed until run time - process designer does not even have to know who should do it or what skills should he or she represent. Developed workflow prototype supports this pattern explicitly, by using activity attributes. All that needs to be defined at design time is the name of an attribute that will point to relevant performer at runtime. Value of this variable can be set, for example, during execution of preceding activity, like in the following example.

```
create activity {
    name "A";
    ...
    following_activities {
        bag { B }
    }
} as A;

create activity definition {
    name "BDef";
    performer {
        Employee where
        ((attributes.attribute where name = "performer").value)
        in name;
    }
```

```
} as BDef;

create activity {
    definition BDef;
    ...
} as B;
```

```
<activity>
  <id>A</id>
  ...
  <following-activities>
    <activity>
      <id>B</id>
      <definition>
        <performer>
          <attribute>performer</attribute>
        </performer>
        ...
      </definition>
      ...
    </activity>
  </following-activities>
</activity>
```

In the above process fragment, automatic activity A is responsible for determining who will perform activity B. Then, upon commencement of task B, resource management module will try to assign an appropriate resource to it, according to the value of attribute `performer`.

Capability-Based Allocation Ability to allocate resources directly, by role, or even to defer work item assignment until runtime, does not suffice in some cases. Let's say that we want to confine performers of task "MakeCreditDecision" only to people that have at least 10 years of experience, and at most 60 years of age. It would be hard to select resource that would match those criteria by applying mentioned patterns. Eligible resources could belong to different groups or organizational units, and hence it is not feasible to assign activity to such performers only by role. Capability based allocation pattern facilitates constructing such criteria at design time. Then, at process execution time, those rules would be evaluated and task would be assigned to conforming resources. This workflow pattern is also supported and can be expressed in the extended SBQL. In order to obtain described scenario, we would have to produce the following SBQL code:

```
create activity definition {
    performer {
        Employee where (experience >= 10 and age <= 60);
    }
} as MakeCreditDecisionDef;
```

Or, analogously, the following XML code in the developed prototype notation:

```
<activity>
  <id>MakeCreditDecision</id>
  <definition>
    <performer>
      <capabilities>
        <capability>
          <capability-name>experience</capability-name>
          <capability-value>10</capability-value>
          <capability-relation>GE</capability-relation>
        </capability>
        <capability>
          <capability-name>age</capability-name>
          <capability-value>60</capability-value>
          <capability-relation>LE</capability-relation>
        </capability>
      </capabilities>
    </performer>
    ...
  </definition>
  ...
</activity>
```

Random Allocation A certain group of workflow patterns has been identified as helpful in assigning resources to tasks. Those are specific rather to resource management implementation, than to notation. First pattern of this group that will be discussed is called Random Allocation. As the name suggests, according to this resource allocation algorithm, performers are given activities on a random basis, despite the amount of tasks that they are currently performing or tasks that await in their individual queues. This allocation algorithm has been implemented in the developed workflow management solution, as one of several methods for performer allocation. In order to use Random Allocation, we have to specify this allocation algorithm in the process definition, like in the following example:

```
create process definition {
    ...
    allocation "RANDOM";
} as PDef;
```

In the case of the XML-based implementation, the process definition should contain the following code:

```
<process>
  <definition>
    <resource-allocation>random</resource-allocation>
    ...
  </definition>
  ...
</process>
```

The resource manager module will read this attribute and allocate resources to tasks within this process according to the specified method.

Shortest Queue Another kind of performer allocation has been identified as the Shortest Queue workflow resource pattern. Here, decision who will be assigned to carry out given task, depends on the number of activities that have already been allocated to this resource. Performer who has the least number of work items allocated will be the first candidate to execute tasks distributed this way. Even when there are several resources eligible for execution of particular work items, the least loaded will be chosen first. Similarly to the previous pattern, choice of Shortest Queue method will have to be specified in the `allocation` section of the process definition:

```
create process definition {
    ...
    allocation "LOAD_BALANCE";
} as PDef;
```

Similarly, the `<resource-allocation>` section is responsible for the same in the XML notation:

```
<process>
  <definition>
    <resource-allocation>load-balance</resource-allocation>
    ...
  </definition>
  ...
</process>
```

Case Handling Some situations in business process management require each process instance to be executed by exactly one resource, i.e. all tasks within such processes need to be assigned to the same performer. This method of work items distribution can speed up running activities, especially when allocated resource needs to have knowledge about process history or progress. This construct is called Case Handling, and has also been implemented in the prototype workflow system. It can be applied analogously to other patterns from this group:

```
create process definition {
    ...
    allocation "CASE_HANDLING";
} as PDef;
```

```
<process>
  <definition>
    <resource-allocation>case-handling</resource-allocation>
    ...
  </definition>
  ...
</process>
```

Upon assignment of resources to work items of this process, the engine will first look at previous activities' performers and reallocate them to this task. If this is the first activity within the whole process, resource will be chosen randomly from the group of eligible performers.

Commencement on Creation / Allocation Delays that occur during process execution usually result from waiting time after particular work item is created and before it is enacted. In some circumstances there is a need to start execution immediately after task is created, without unnecessarily waiting for performer allocation or waiting in the resource's work queue. That is why resources should be allocated to those activities almost at the time of creation, so that commencement could begin instantly. Described construct has been identified as the Commencement on Creation pattern. It is naturally supported by the developed workflow engine - all activities are, by default, executed immediately, as long as resources suffice. Moreover, the XML notation enables process designer to determine specific resource that will be assigned to this particular task at design time. Hence, those tasks will be eligible for execution immediately after creation.

There has also been identified a pattern that is closely related to Commencement on Creation and is called Commencement on Allocation. The only difference is that in the latter case, work items can be created, but cannot be enacted until they are assigned to appropriate resources. After that, commencement begins promptly. Workflow engine and notation that have been developed within this work also support this construct. Process manager module invokes relevant methods of the resource manager, which results in performer allocation. Then, activity is available for instant execution.

Simultaneous Execution Business processes in the real world tend to be long-running because of many factors, like involving data exchange with external systems, waiting for customer's response, or because of resources shortage. When particular performer starts carrying out a task and then waits for some events to occur, he or she may meanwhile work on another task or case. By starting several activities at a time, there is a greater probability that all of them will be finished earlier and working time would be utilized the best way. Otherwise, each task would be executed sequentially, effectively decreasing total workflow throughput. This notion of assigning more that one activity at a time to a single resource is called Simultaneous Execution. It is supported by the developed workflow solution, as it does not limit particular resource's task queue size. Thus, several work items may be assigned to single performer, fostering parallelization of work, and hence boosting overall workflow throughput. The following SBQL and XML examples depict construct described above:

```
create activity definition {
    ...
    performer { Employee where name = "JohnDoe01"; }
} as ADef;

create activity definition {
    ...
    performer { Employee where name = "JohnDoe01"; }
} as BDef;

create activity {
    definition ADef;
    ...
} as A;

create activity {
    definition BDef;
    ...
} as B;

create process definition {
    activities {
```

```
        bag { A, B }
    }
    ...
} as PDef;
```

```
<activity>
  <id>A</id>
  <performer>JohnDoe01</performer>
  <following-activities>
    <activity>
      <id>B</id>
      <performer>JohnDoe01</performer>
      ...
    </activity>
  </following-activities>
  ...
</activity>
```

This simple process definition assigns two tasks - A and B - to the same performer - JohnDoe01. Upon execution of this process, workflow engine will allocate both activities to the same resource, so that A, as well as B will be on JohnDoe01's work list, eligible for simultaneous enactment.

Automatic Execution Even in real life processes, not all tasks are accomplished by real people. Many of the activities are required to be run automatically, for example by launching external procedures, or performing some heavy calculations related to business logic. The automatic execution pattern is a natural complement to all the mechanisms that assume human presence during realization. Proposed SBQL workflow extension requires use of the `work` clause to specify behavior, whereas the implemented form of this pattern enables process designers to attach Java classes to particular activities. Additionally, those tasks need to have their `performer` attribute set to `AUTO`, as it is shown below.

```
create activity definition {
    work {
        /* SBQL code to be executed */
    }
    performer { "AUTO"; }
} as ADef;
```

```
<activity>
  <id>A</id>
  <definition>
    <id>A-definition</id>
    <class>CalculateDiscount</class>
  </definition>
  <performer>AUTO</performer>
  ...
</activity>
```

The former code snippet will make the process engine execute code enclosed within the `work` statement, and in the latter example, process management engine will execute the procedural part by looking for the Java class named `CalculateDiscount`, then by instantiating it and invoking appropriate method thereof.

3.3 Data patterns

Much effort has also been put in identifying common constructs related to data handling within workflow management. Those patterns involve mechanisms for storing, transferring and accessing data, as well as dynamically selecting process paths according to activity or process-scoped information. All of the data patterns described below are explicitly or implicitly supported by the workflow management system developed within this work.

Scoping and Accessing The first group of data patterns that will be discussed, are common constructs related to data visibility during process execution. They are rather tools than patterns themselves, as they only constitute some foundations for dynamic data-based routing or communication with external systems. The developed workflow engine and notation explicitly support two scopes, namely: **Task Data** and **Case Data**. The first one is very simple and may be used to store local information like calculation outcomes, or temporary task data. Going further to the second pattern, designers can define attributes that span the whole process instance, i.e. are accessible from any of tasks that make up this business process. This approach can be useful when one wants to publish information belonging to one single activity, to the whole process. In order to store or manipulate data visible within those two scopes, the process designer needs to define task-level attribute for Task Data or process-level attribute for Case Data. Both of these scenarios have been illustrated below.

Task Data:

```
create activity definition {
    attributes {
        attr1 : string;
    }
    ...
} as ADef;
```

```
<activity>
  <id>A</id>
  <definition>
    <id>A-def</id>
    <attributes>
      <attribute
        name="attr1"
        class="java.lang.String"
        comment="Activity scoped attribute"
      />
    </attributes>
  </definition>
  ...
</activity>
```

Case Data:

```
create process definition {
    attributes {
        attr2 : integer;
    }
    ...
} as PDef;
```

```
<process>
  <id>P</id>
  <definition>
    <id>P-def</id>
    <attributes>
      <attribute
        name="attr2"
        class="java.lang.Integer"
        comment="Process scoped attribute"
      />
    </attributes>
  </definition>
  ...
</process>
```

Apart from those two directly supported constructs, the developed process management system also supports one more workflow pattern related to data visibility, but on the implicit basis. Subprocesses nested inside other processes constitute a form of scope other than just task or case. Infinitely nestable processes provide subprocess data scope, which in fact can span any number of tasks or subprocesses nested within. This indirectly supported pattern is called **Scope Data**.

Data elements described in one of the above scopes needs to be accessed either by the same activity, other activity from the enclosing process or even an external system. Those attributes can then be used in further processing (eg. calculations), transformed, or passed further to another system. Hence, three similar workflow data patterns can be identified here - **Task to task**, **Block Task to Sub-Workflow Decomposition**, and **Sub-Workflow Decomposition to Block Task**. While the first one involves accessing activity's attributes from another task of the same process, the second and third construct implementations take advantage of nested subprocesses in order to enable two way data access between the task which contains subprocess, and all the activities that constitute this subprocess.

Data-based conditions and routing The workflow management concept and implementation that has been developed within this work greatly emphasizes flexibility and dynamic process path selection during process execution. Hence, patterns involving various decisions depending on data existence or value, have been implemented thoroughly as many other features hinge upon their functionality. The **Task Precondition** pattern is probably the most viable and the most frequently used construct among all workflow data patterns. By applying it, process designer can model other important constructs from the Control Flow group, like N-out-of-M Join or Arbitrary Cycles. The following SBQL and XML examples show a typical usage of Task Precondition in order to select one of three alternative paths:

```
create activity {
    name "B";
    preconditions {
        ((Activity where name = "A").attributes.attribute where
        name = "credit_amount").value < 10000;
    }
} as B;

create activity {
    name "C";
    preconditions {
        ((Activity where name = "A").attributes.attribute where
        name = "credit_amount").value between 10000 and 1000000;
    }
} as C;

create activity {
    name "D";
    preconditions {
        ((Activity where name = "A").attributes.attribute where
        name = "credit_amount").value > 1000000;
    }
} as D;
```

```
<activity>
  <id>B</id>
  <preconditions>
    <condition>
      <attribute>
        <activity>A</activity>
        <name>credit_amount</name>
        <value>10000</value>
        <relation>LT</relation>    <!-- less than -->
      </attribute>
    </condition>
  </preconditions>
</activity>

<activity>
  <id>C</id>
  <preconditions>
    <condition>
      <and>
        <condition>
          <attribute>
            <activity>A</activity>
            <name>credit_amount</name>
            <value>10000</value>
            <relation>GE</relation>    <!-- greater or equal -->
          </attribute>
        </condition>
        <condition>
          <attribute>
            <activity>A</activity>
            <name>credit_amount</name>
            <value>1000000</value>
```

```
            <relation>LE</relation>    <!-- less or equal -->
          </attribute>
        </condition>
      </and>
    </condition>
  </preconditions>
</activity>

<activity>
  <id>D</id>
  <preconditions>
    <condition>
      <attribute>
        <activity>A</activity>
        <name>credit_amount</name>
        <value>1000000</value>
        <relation>GT</relation>    <!-- greater than -->
      </attribute>
    </condition>
  </preconditions>
</activity>
```

Those three activities represent consecutive process paths, from which only one has to be chosen. The choice depends on the value of `credit_amount` attribute set during execution of preceding task. For small amounts, manager's acceptance is not needed, whereas in the second case (for amounts between $10,000 and $1,000,000) - manager's acceptance is required. There is also a third case, where credit amount is greater than $1,000,000. Then, a managing director needs to confirm credit. As we can see, no two paths can be selected simultaneously, because of exclusive preconditions for each of the subsequent activities.

Similar construct is the **Task Postcondition**. Unlike Task Precondition, this one introduces condition check after particular activity has completed. It can be useful in situations when the process has to wait until specific task- or process-scoped attribute is set or equal to some predefined value. The following sample scenario can illustrate one of these situations:

```
create activity definition {
    name "PrepareAnnualSalesReportDef";
    attributes {
        calculated_data : integer;
    }
} as PrepareAnnualSalesReportDef;

create activity {
    name "PrepareAnnualSalesReport";
    definition PrepareAnnualSalesReportDef;
    postconditions {
        waitfor {
            ((Activity where name = "PrepareAnnualSalesReport")
            .attributes.attribute where
            name = "calculated_data").value <> null;
        }
    }
}
```

```
<activity>
  <id>PrepareAnnualSalesReport</id>
  <definition>
     <id>PrepareAnnualSalesRepostDefinition</id>
     <attributes>
      <attribute
        name="calculated_data"
        class="java.lang.Integer"
        comment="Container for data calculated during execution"
      />
    </attributes>
  </definition>
  <postconditions>
    <condition wait="true">
      <attribute>
        <activity>PrepareAnnualSalesReport</activity>
        <name>calculated_data</name>
        <relation>NN</relation>    <!-- not null -->
      </attribute>
    </condition>
  </postconditions>
  ...
</activity>
```

Here, task named `PrepareAnnualSalesReport` only coordinates sales report preparation, but the actual processing and calculations are delegated to some external subsystem, whose role is to do some heavy processing and put results in an appropriate task attribute. Therefore, this activity will finish immediately upon reception of that externally calculated data.

Sometimes a need arises to start execution of specific task upon some event, usually triggered by data changes. This scenario can be initiated from the outside (i.e. by external system), as well as from the inside - by other activities or processes running inside the same workflow engine instance. The situation described above is in fact a special case of the Task Precondition pattern, called **Task Trigger**. Representation of this construct in the developed XML notation is straight forward. Like it was mentioned above, it can be modeled by using task precondition with `wait` parameter set to `true`, so that the process management engine could wait for the triggering event to occur. Behavior of the Task Trigger data pattern can be represented by the example SBQL code shown below.

```
create activity {
    name "OrderNewItems";
    preconditions {
        waitfor {
            (attributes.attribute where name = itemCount).value < 10;
        }
    }
} as OrderNewItems;
```

Or, with the following XML markup:

```
<activity>
  <id>OrderNewItems</id>
  <preconditions>
    <condition wait="true">
      <attribute>
        <name>itemCount</name>
        <value>10</value>
        <relation>LT</relation>    <!-- less than -->
      </attribute>
    </condition>
  </preconditions>
  ...
</activity>
```

Here, the activity named `OrderNewItems` will be launched immediately after the value of `itemCount` attribute falls below 10.

The last workflow data pattern that will be discussed is **Data-based Routing**. Being a kind of a generalization, it depicts behavior of several previously described constructs. It is defined as the ability to alter business process flow during its execution, as a result of data events. As it has been proven in previous paragraphs, the designed workflow notation and management system fully supports data patterns involving dynamic process changes - Task Precondition, Task Postcondition and Task Trigger. Thus, being general and flexible by design, it also supports the broad form of those constructs - the Data-based Routing pattern.

4 Features and capabilities of the developed workflow management solution

Today's workflow management systems are usually based on traditional approaches to this subject, for example theoretical foundations like Petri nets. Those systems have severely limited flexibility, when it comes to performing distributed and truly parallel tasks. Processes managed that way are not only limited to available resources, but also to keeping all activities in sequences. However, many of those activities can be successfully parallelized, as far as processing is concerned. The only bottleneck in this model are available resources or lack of them. This concept is much closer to reality than sticking tightly to sequential alignment of activities that should otherwise be performed in parallel. The developed workflow management solution is an attempt to go beyond the sequential processing model and to parallelize process execution as much as possible. Notation developed for this prototype supports the above concepts - if not specified otherwise, all activities are carried out in parallel. Another invaluable feature is capability of nesting workflow processes. Such nested processes do not have to "know" about each other, hence they can be infinitely nested without any notable performance overhead or scalability loss. During the design of data structures for business processes, much emphasis has been put on generality, which benefited in wide applicability and flexibility of process specification. Even very complex business processes can be stored and executed using relatively simple and intuitive XML syntax. Every construct related to routing of processes can be specified by using pre- and postconditions. These conditions can depend on many factors, such as absolute date and time, delay after the preceding activity or process has ended, or even according to data contained in particular process or activity. The prototype also enables users to specify a condition as a Java class that implements well-defined interface. At runtime, those classes are executed and the outcome is checked for true or false. As it was stated above, process execution depends on availability of required resources. In this workflow solution, resource allocation has also been considered, as being crucial to proper business process handling. Resources are divided into two main groups: human and non-human. While non-human resource allocation is achieved simply by specifying which resources are needed by particular activities, the human resources allocation is a bit more complicated. Module responsible for this enables assigning resources to roles, adding capabilities to particular resources and during the process execution, dynamic allocation of human resources according to roles, skills or other characteristics (i.e. experience, age, and so forth). There are also available three algorithms for distribution of tasks among available resources: load balancing (assigning tasks to resources that have the minimum number of tasks assigned), random (assigning tasks to one of the eligible resources randomly), and case handling (for particular workflow process, tasks are assigned to the same resource, if not specified otherwise). Workflow management systems should integrate easily with developed information systems, providing all the features in the form of clear and concise API. The developed prototype system is no exception to this rule. It not only exposes public interfaces to manipulate processes or resources, but also enables the external system to execute XPath queries directly on the underlying workflow data. Using this feature, one can perform simple queries like list of given user's tasks or even achieve more advanced functionality like reporting. Failures or exceptional situations during execution of business processes occur very often in the real world and to reflect this, the solution also has support for error handling and compensation. Process designer's task is to define special Java classes and assign exceptions to them. Everything else is managed by the Process manager module - in case of an error that designer or developer specified, the special compensation procedure is run, which reverts the effects of erroneous process. Another important issue in workflow management systems is data handling. Among numerous commercial systems, data can be attached to individual activities, processes or even instances of the management systems. Sometimes, the data can even be transfered to other processes or external systems. Approach to data handling in the developed

workflow system has been kept quite simple and general, yet flexible and powerful. Data can not only be assigned to particular process instance or activity, but it can be manipulated by automatic tasks carried out as Java classes. What is more, pre- or postcondition checks can depend on data contained in any part of the process, enabling system architects to design efficient and flexible data-based routing.

4.1 Technologies and tools used

Numerous tools and technologies have been used to develop this workflow solution. Starting from J2EE standards implementation, through various XML-related technologies, unit testing frameworks and ending with build automation and dependency management tools. All of those have been chosen according to the needs of the project. None of them has been applied without profound consideration and together they constitute a framework that greatly simplifies development and promotes sound design principles.

J2EE standards The most fundamental decision was whether to use J2EE application server or develop a stand-alone Java application framework that does not need to run within application server environment. The main factors that influenced this decision was distribution of process execution and inherent thread safety of operations carried out within J2EE application server. Another benefit of using J2EE application server was declarative transaction demarcation (Container Managed Transactions) and declarative security managed by the server. Two types of Enterprise JavaBeans have been used, namely: stateless session beans and message driven beans. Although the session beans are used only as adapters between remote clients and POJO (Plain Old Java Objects) service implementation and for transaction demarcation, the application of message driven beans with JMS (Java Message System) has profound implications and fundamental meaning for the developed solution.

XML Schema This standard of XML documents syntax control has been established by the World Wide Web Consortium and has proven its value in numerous professional application deployments. It enables exchange of data in a well-defined form between homogeneous or heterogeneous systems. As a standard, it is vendor independent, so various implementations on different platforms should exchange data easily and without additional effort put into conversion of documents. In this workflow management solution, XML Schema has been applied to restrict correct form of process, activity, as well as other entities used along the system. Data in this system is stored in XML format, so XML Schema enforces syntactic rules while writing or parsing documents.

XMLBeans XML Schema with XML would not retain its full power without delivering its strong typing to the Java world. XMLBeans developed by Apache Software Foundation (created by BEA Systems and donated to ASF) is an open-source, easy to use framework for generating Java classes from XML Schema documents. This enables developers to write code and create structures that can be easily serialized to an XML format or the other way around - parse XML files into Java objects containing appropriate data. This library also enables developers to use XPath queries on Java objects in order to retrieve needed data. The workflow management system uses XMLBeans extensively - to create, manipulate and search for objects containing process, activity and resource information. It is the foundation of the Process Manager module.

JUnit Quality software, likely to change along the development process, requires some assurance that previously implemented features will not be broken by those added more recently. In small applications, every feature can be easily tested by running some examples by hand, however as the code gets more

and more complex, testing by hand can get tedious, error-prone, and it can take very much time. Even then one can't be sure that every possible case has been tested. Here is where automated unit testing tools do their job. JUnit is the most popular and the most widely-used unit testing framework for Java. It is simple, yet very powerful. All the developer has to do is write simple testing method (a test case) with assertions for each case from the application code to be tested. Then, all unit tests can be run automatically, assuring that everything works correctly. JUnit has been used in this project to test operations performed by main modules of the application. It integrates well with existing build automation tools (like Maven) enabling tests to be run automatically with each build.

Maven 2 It is hard to maintain all resources in most of serious projects developed nowadays - compile source files, generate appropriate descriptors, run automatic tests, put necessary resources as well as compiled class files into packages. This takes much time to do by hand, can be error-prone and in the long run very frustrating. Maven 2 is a second, enhanced and improved version of popular Maven build tool. Maven not only manages all files in a project, but also takes care of all required dependencies (e.g. libraries needed to build application). It also integrates well with existing build tools (like Apache Ant) and IDEs (like Eclipse) enabling developers to boost development and deployment.

4.2 Two approaches to workflow management

Two disjoint implementations have been developed. First one is a thread-based, stand-alone library that can be run with a simple Java application, without imposing any design decisions and with no further implications on the application architecture. This implementation is not distributed, does not use transactions and can be run only within single virtual machine. It has been used mostly as an early prototype and implementation of conceptual foundations for the second prototype, and thus has not been included with this work. The second one is more complex and requires some additional infrastructure. The applications that use it must be run within a J2EE application server, because modules of the workflow management system are implemented as EJB components and use transactions managed by the container. This approach is mainly based on asynchronous message processing - messages are emitted by JMS client and they are consumed by message-driven EJB components. This architecture greatly simplified code related to concurrent processing that would otherwise be hard to debug and maintain. What is more, EJB components deployed in the container gain transactional behavior, security and many other services done by the container. With this approach, distributed processing can be achieved very easily and transparently to developers of applications that use this workflow management solution.

4.3 Process management module and resource management module

Main components of the project are the process and resource management modules. First of them enables external system's programmers to create, launch and manipulate workflow processes, as well as single activities that those processes consist of. It takes care of the data passed along the process, pre- and postconditions checks, error and compensation handling, and thus, the whole logic related to workflow routing. Any serious workflow solution could not exist without considering resources, hence a module responsible for resource management has also been developed in this project. It enables the process to allocate just any kind of resources - people, computers, rooms, equipment, etc., in a storage-independent manner. What is more, applications that will use this module, will have access to user and resource management operations, such as creating new users, users' role and capability management or available resources management and monitoring.

4.4 Running an example process

Together with the created workflow management engine, a windows-based application has also been developed in order to launch and visualize processes. Two example processes have been created: the discussed in previous chapters Insurance Claims Handling process, and the Organize Trip process that is likely to be carried out by a travel agency. Execution of those processes can be achieved by running the supplied Workflow GUI application, and selecting appropriate process from the drop-down list, as shown in Fig. 23.

Figure 23: Workflow GUI Application

From this window, one can load a process, so that it will be shown in the form of a graph. Now, the selected process can be launched by clicking the "Launch Process" button. The screen shot in Fig. 24 shows graph of the Insurance Claims Handling process.

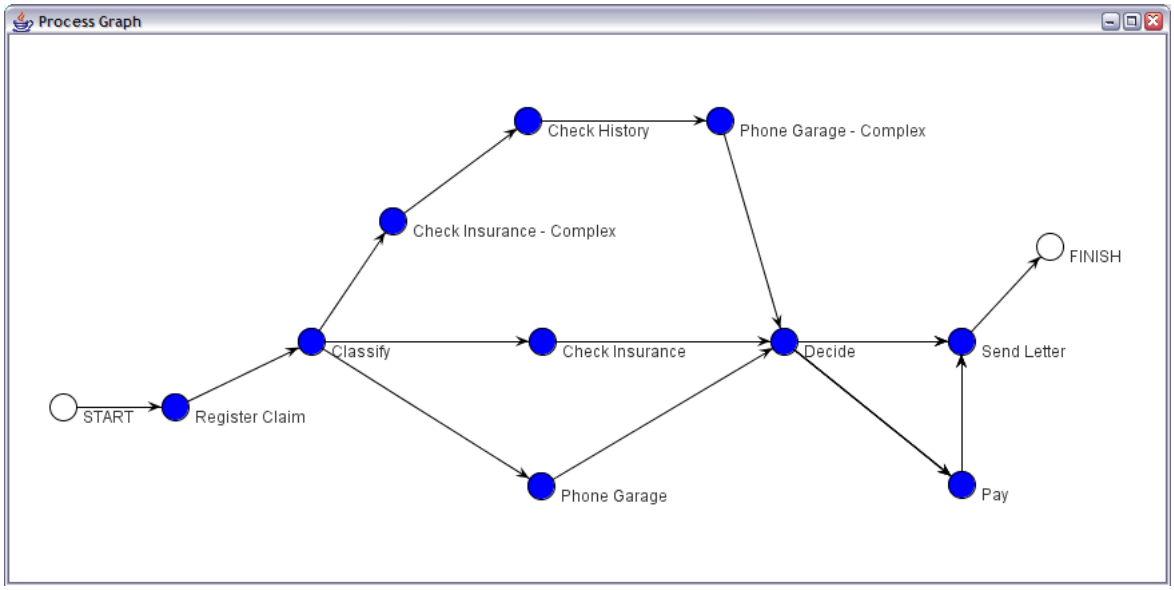

Figure 24: Example process graph

Activities are shown as filled circles, whose interior color depends on the current status. There are two additional vertices in each generated graph, named START and FINISH, which are not real activities, but have been added for clarity and comprehensibility purposes. After clicking the "Launch Process" button, vertices' color changes can be observed, which means that the activities are being carried out. In the Insurance Claims Handling process, the first path selection is done within the "Classify" activity. Its status is set to NOTIFIED by the process management engine, which means that at this point, the process is waiting for user's decision. By clicking vertex representing this activity with the left mouse button, user can choose appropriate value for an attribute from the window shown in Fig. 25.

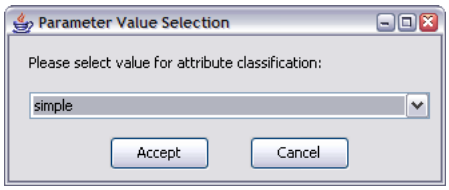

Figure 25: Attribute selection window

Once the "Accept" button is clicked, the workflow engine will choose relevant process path according to the following activities' preconditions. The "Decide" task, on the other hand, is the point in the workflow process, where previously split paths converge into the single execution route. This is another task, where user is responsible for selecting attribute's value and thus altering the process flow - the "Pay" activity will be carried out only if the "decision" attribute has been set to "true".

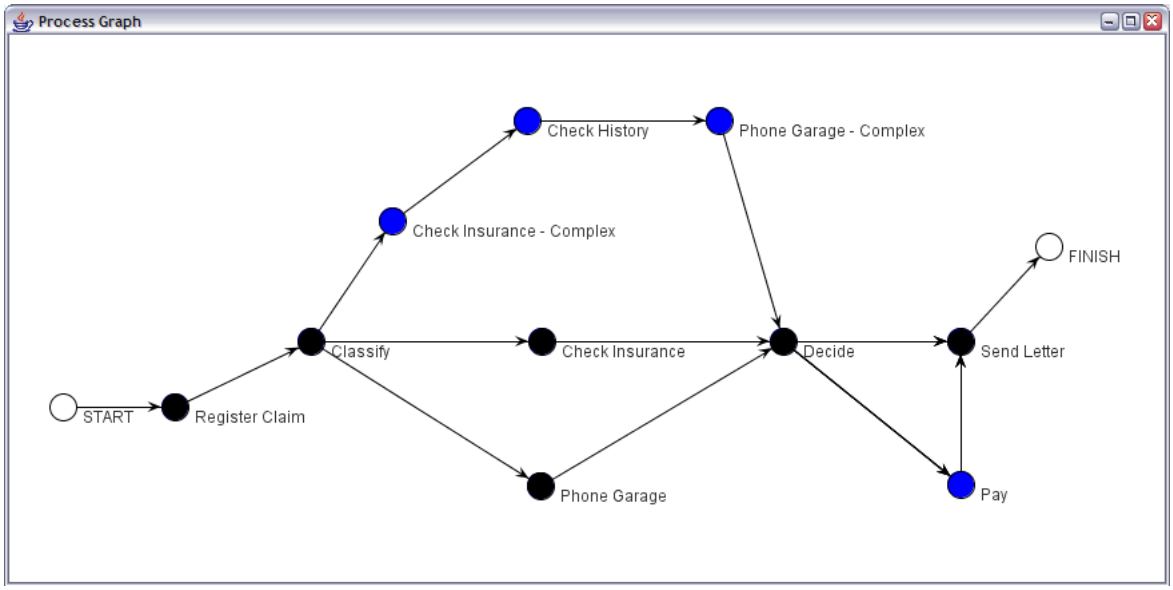

Figure 26: Finished process

The process graph after execution is presented in Fig. 26. Note that in this particular case, the claim has been classified as simple, and later on it has been rejected (the "Pay" activity has not been executed). The mapping between activity statuses and colors is shown in Fig. 27. The difference between "Finished" and "Completed" statuses is that activities with the former status have successfully finished their procedural part and at that moment, their postconditions are being evaluated. The latter one means that an activity has completed execution, and its postconditions have been evaluated to true, enabling the workflow to proceed to another tasks.

Figure 27: Activity statuses and their respective colors

5 Design decisions justification and description of the architecture

5.1 General description of implemented architecture

Main goals that we were trying to achieve during development of this project was generic approach to workflow management, extensibility, reliability, portability by design and lightweight architecture. The most important components of this system, namely the Process Manager and Resource Manager were designed as loosely coupled and lightweight modules. Both of these services have been developed as POJO services wrapped with session EJBs, which greatly simplified unit testing outside of the container - session EJBs are just adapters that invoke services' methods. Communication is done in an asynchronous manner, which means that workflow process execution can be as much parallelized as possible. Key technologies used to develop this messaging infrastructure were Java Message System and Message Driven Beans. Although this is a lightweight approach, it's also very reliable thanks to inherent thread-safety of Enterprise JavaBeans running inside the container. Declarative transaction demarcation is another value added by the container and leveraged in this project. Declared transactions are at runtime managed by the application server, which means that every method in workflow management module, as well as resource management module is executed in an atomic manner. Yet another benefit of using J2EE container is declarative security - all the developer has to do is declare roles and principals eligible to perform specific operations and the actual control and security checks are managed by the server.

5.2 Detailed information about the developed components

Process management module

Central point of all operations related to process execution is the process management module. It has been implemented as a POJO service - simple Java object that exposes public methods for manipulating processes and activities. It then has been wrapped with a stateless session bean for security, transaction management and thread safety purposes, as stated in the above section. Process management module accesses data access object, which in turn provides methods for direct manipulation of data. Execution of a process is started by invoking the appropriate method from the process management module's interface, and during the execution, asynchronous messages (JMS) are sent to appropriate consumers, which are Message Driven Beans. First one is an Activity Executor - once it receives message containing data about activity to run, it executes it immediately by instantiating relevant class and invoking methods from its well-defined interface. As particular task completes its execution, another asynchronous message is sent, but this time to the Activity Observer message driven bean. Upon reception of this message, the observer reacts to event contained in it - typically just an activity status change notification, and sometimes an error message. There can be many EJB MDB components acting as Activity Executors and Activity Observers, possibly distributed throughout the network, however this distribution would be transparent to the message producers. This approach greatly increases scalability of potentially distributed workflow processing, while thread safety and transactional behavior is retained. The sequence diagram in Fig. 29 illustrates actions described above in more detail.

This module provides a coarse-grained interface for the client applications that use it, enabling easy integration and hiding from the users all the implementation details. The most significant operations contained in this interface have been listed below.

- create process instance
- launch process

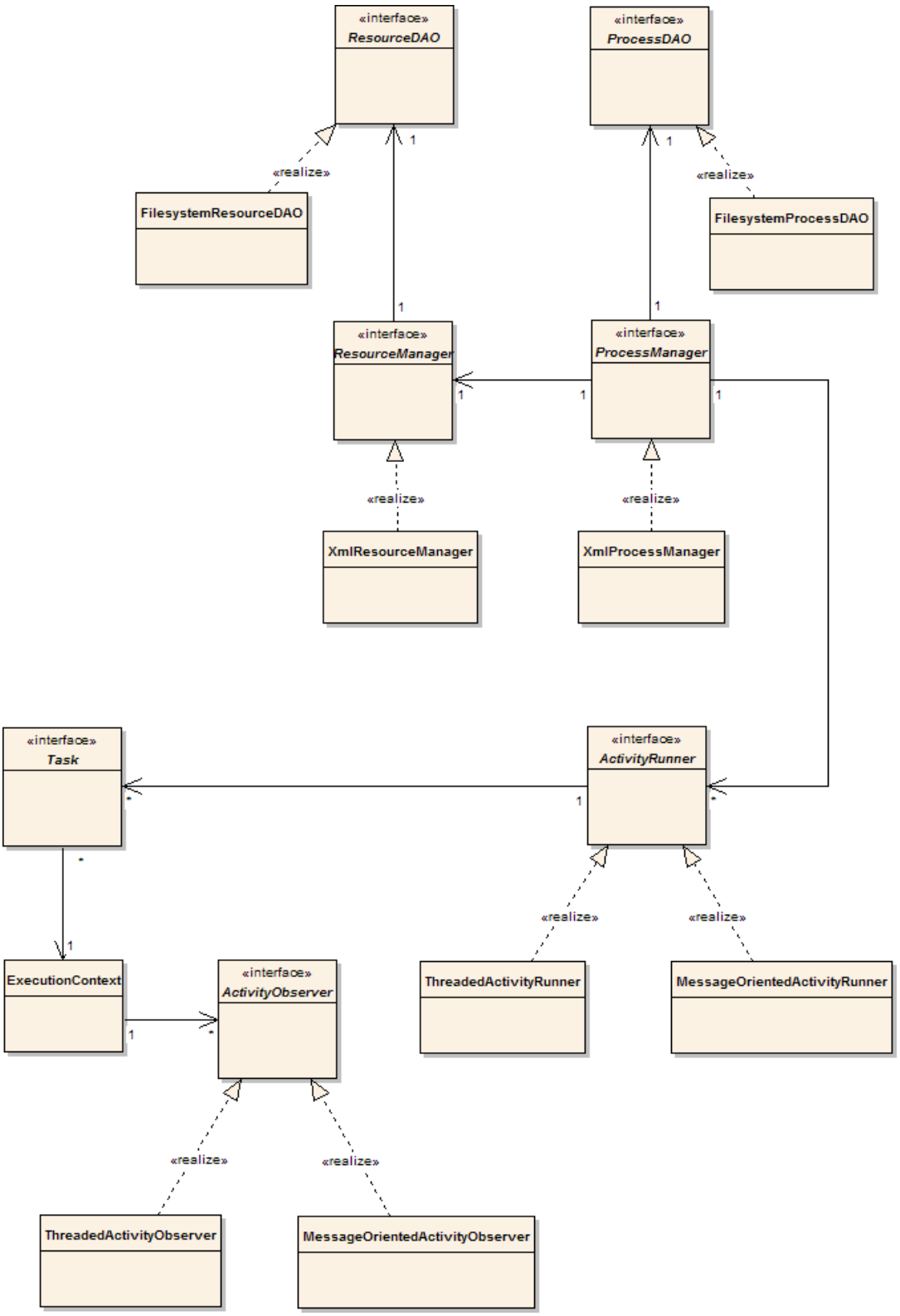

Figure 28: General view of the system's architecture

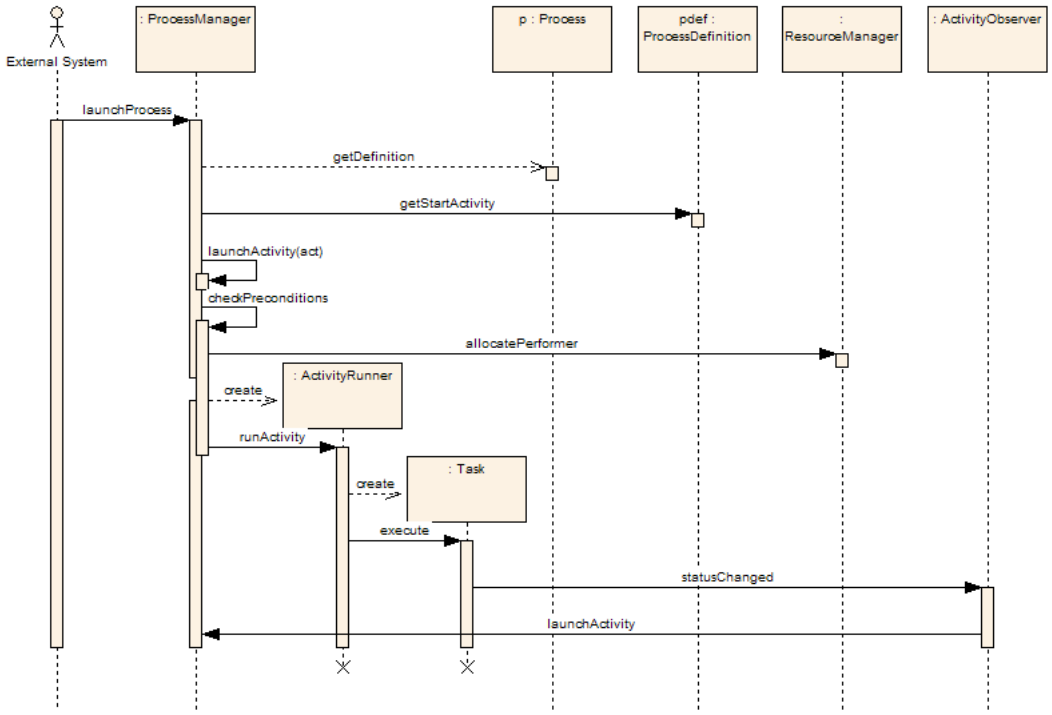

Figure 29: Launching process instance

- create/retrieve/update process attribute
- create/retrieve/update activity attribute

For more detailed specification of process manager's interface, please refer to Appendix C.

Resource management module

Another service object that has been implemented in the prototype, is the resource management module. For consistent architecture and clear design, this module has also been developed as a simple Java object, exposing appropriate methods for manipulating resources. Again, there has been created a stateless session bean, dedicated to wrapping this POJO, and hence, providing transaction handling, declarative security, and thread safety. This component would be deployable and executable inside J2EE container. Unlike the process management module, this service does not utilize distributed messaging, such as JMS - it only accesses relevant data access object and by using it, performs all the data manipulation operations. This module's interface provides several useful methods that can be invoked from within process management module, as well as by this module's potentially remote clients. Noteworthy methods from the former group are as follows:

- allocate resources to activity
- allocate performer to activity
- deallocate resources from activity

The public interface exposed to other clients, that are not a part of this workflow management system, limits functionality to only those operations that are not invoked from within process management module. It contains methods that provide the following functionality:

- create/retrieve/update/remove user
- create/retrieve/update/remove role
- create/retrieve/update/remove resource

Detailed description of those components' API, together with legal parameters, returned values, and thrown exceptions, has been placed in Appendix C.

5.3 Description of applied design patterns

As a design pattern, we understand a common solution to recurring problem. First design patterns appeared as an architectural term, originally coined by Christopher Alexander. Since then, pattern language has been widely adopted in computer science, especially in object oriented design and development. They not only provide solutions to design problems, but constitute a kind of common language that helps easily exchange concepts between designers or architects. In our workflow solution, we have also applied several object oriented design patterns that simplify development and promote code reuse.

Observer Typical graphical user interface libraries or frameworks, especially those written in Java, use the event-driven model. It means that a particular action taken by the user, triggers some procedures that respond to that action. The user action is called an event, and that special procedure is an event handler. The Observer design pattern is conceptually equal to those briefly described solutions in case of user interfaces, however it is more general and does not impose any concrete technologies or implementation strategies. In this workflow system, two observers have been created - first one reacts to new processes that are ready to be executed, and the second one is a process event handler, i.e. it handles changes that occur during the process execution. These changes may include process status changes or errors that happen during execution. Application of this pattern in the workflow management system is not a straight object oriented observer pattern as described in [Gamma et al.], because it leverages J2EE standards such as Java Messaging System (for asynchronous message exchange and queuing) and Message Driven Beans (for processing of those messages).

Factory Method Another very useful pattern is the Factory Method. It helps to defer class instantiation until runtime - before that moment, application doesn't even know that those classes exist. Typical implementations of this pattern involve mapping between text strings and Java classes. Sometimes that mapping is done inside Java structures like HashMap or Hashtable, and sometimes it is extracted out of code to external text file. The latter solution has been applied in this project. It is used for instantiating classes that implement the Data Access Objects pattern (explained below) containing persistence logic of processes and resources. This implementation slightly differs from the typical one - it leverages Java 5.0 generics feature, making Factory even more robust, type-safe and hence, less error-prone.

Singleton During application design or implementation, often arises the need for an object with only one instance, and with single point of access. Objects satisfying those requirements are called, naturally enough, singletons. Their most common use is to act as a factory producing particular objects or perform some logic related to lookups of remote or local EJB objects in the JNDI directory. Application of singleton pattern in this system represents the latter case. Although true singleton pattern cannot be

achieved in J2EE environment due to distribution and usage of multiple virtual machines, singleton implementation developed in this system is satisfiable and meets all the requirements.

Command There are times that designers or developers don't know how particular classes should behave, and so they want to defer this decision until late implementation or even after that phase. Moreover, there may be several classes with different behavior, but conforming to the same interface that is known at design time, that would be applied interchangeably. This is known as the command design pattern. One has to declare an interface, and the calling class will invoke methods defined in it. Implementation of this interface can be even supplied at runtime (provided that appropriate reflection instructions are executed to load the class). In this project, the command design pattern has been applied to execute particular activity's work units as Java class files supplied in the workflow process definition in case of activities to be carried out automatically. Class implementing mentioned interface is instantiated and provided to the activity executor component for immediate execution.

Data Access Object This pattern is in fact a special case of the Strategy design pattern, as described in [Gamma et al.]. It encapsulates all persistence logic in one object, making the service layer independent from the underlying data storage. If the data store changes, all the developers have to do is reimplement the DAO interface. Although the underlying database will change in extremely rare cases in the commercial environment, it is good for the design to retain its portability across database platforms. The developed workflow management solution uses Data Access Objects to deal with persistence logic of processes and resources. Those objects are persisted to the filesystem in a form of XML documents, but as has been stated above, design is portable so it can be easily ported to another persistence technology like relational or object oriented database.

Service Locator Service Locator is not a typical pattern implemented in all Java applications. Its main goal was to separate code that is used to lookup remote or local objects in the J2EE container environment. By separating this infrastructure-related code and keeping it only in one, central place, code is not unnecessarily duplicated and good practices such as code reuse are promoted. It is quite natural to implement Service Locator as a Singleton-like object (described above). In our project, the Service Locator pattern is applied to look up Process Manager and Resource Manager EJB components in the JNDI directory. Now, instead of invoking several methods and performing numerous type casts, all that needs to be done is encapsulated within one method call.

Session Façade Keeping business logic scattered over several business objects in the EJB tier is not quite a good practice. Possibly remote client must perform calls to many EJB components (including CMP or BMP entity beans), then those components call other beans, and so on, and so forth. Inventors of the Session Façade design pattern advocate centralization of business logic in a few stateless session beans with coarse-grained interfaces. From this central point of the service layer, local calls to other beans are delegated. This pattern greatly simplifies development and improves maintainability of architecture designed this way. The Session Façade has found his place in our workflow project - all business logic related to manipulating processes and activities is centralized in the Process Manager module. Similarly, all code that operates on resources has been centralized in the Resource Manager module.

6 Summary

Workflow and BPM systems are without any doubt one of the fastest developing area in today's software engineering world. Numerous companies produce their own ad-hoc platforms, organizations build standards, and the academic field also comes up with new and innovative workflow solutions.

However, none of the existing concepts, notations or implementations have moved this far towards flexibility and power, as the approach defined and developed within this work did. The idea that we came up with is a natural extension of Stack-Based Approach and thus complements Stack-Based Query Language by adding new functionality dealing with workflows. Being a generic solution by design, and being built on object-oriented concepts, it isn't tied to any of existing technologies or business process notations. This approach deals with some of the problems encountered by traditional workflow systems and solves them in a bit different way, by using alternative assumptions and through different design decisions.

First of those common workflow problems is insufficient parallelization of activities being carried out. Most of the products and notations available on the market assume that every task has to be executed sequentially, after its predecessor completes. In reality, great majority of processes are carried out in parallel, and only in special cases, a high level synchronization mechanism is required to define sequence of activities. This problem is absent in the developed concept, as all the processes and their activities are inherently parallel. What is more, the mentioned synchronization mechanisms allow designers to restrict specific order of tasks in a process, as well as form sophisticated pre- and postconditions, in accordance to which, the appropriate process paths will be selected.

Dynamic process changes is another vital area in the field of workflows, and has also been addressed in this thesis. By increasing flexibility up to this degree, processes defined in the system can reflect real-life business processes in more detail, and hence rapidly adapting to changing circumstances. Resolution to this problem is present in proposed SBA/SBQL extension concept, and thus, also in the Java-based prototype implementation.

The problem of exceptional situations handling has also been tackled with within this work. Errors or other abnormal circumstances need to be dealt with in a declarative manner, so that when such event is encountered, an appropriate process path can be chosen, together with launching special compensation procedure that will leave the system intact, even after serious errors or failures. The developed SBA and SBQL extensions, together with prototype Java implementation, resolve this issue by letting process designers or developers to specify process exception handling in an easy way, using clear and comprehensible notation.

Flexible and dynamic resource allocation, as another recurring issue of workflow systems, is also supported in the proposed extension. The concept assumes a number of ways to allocate a human resource (task performer), as well as non-human resources. However, not much emphasis has been put on this area of developed workflow solution, which leaves an opportunity for more future work on those concepts.

What has been designed and implemented, is a powerful, flexible and extensible approach to business process management. All the goals that were planned, have been achieved. An easy to use, comprehensible and yet powerful notation, independent of any particular standardized or ad-hoc graphical notation, as well as any underlying technology. Nevertheless, there have been many workflow areas that the author found broader than the scope of this thesis, and hence they have been left out for future development. Those regions include security and access control, more sophisticated resource management that would reflect real-life resources with more fidelity, and advanced compensation mechanisms. Altogether, this forms foundations for future implementation work, and further development of the concept itself. Basing on well understood, object-oriented principles, the model can be easily modified, extended and enhanced in multitude of directions.

References

[Subieta] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Warszawa, 2004.

[Aalst] W. van der Aalst, K. van Hee, *Workflow Management: Models, Methods and Systems*, MIT Press, 2002.

[Aalst et al.] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros, *Workflow Patterns*, 2003.

[WfMC] Workflow Management Coalition, *Workflow Terminology & Glossary*, 1999.

[Baeyens] T. Baeyens, *The State of Workflow*, JBoss inc., 2006.

[Momotko] M. Momotko, *Tools for Monitoring Workflow Processes to Support Dynamic Workflow Changes*, PhD thesis, Institute of Computer Science, Polish Academy of Sciences, 2005.

[Gamma et al.] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[Alur et al.] D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, Second Edition, Prentice Hall, 2003.

A Proposed SBA and SBQL Extensions

A.1 Stack-Based Approach Object Store Extensions

Below is the detailed technical description of proposed Stack-Based Approach extensions, consisting of object specification in SBA notation, together with definition of what each element is responsible for. Note that referenced objects (like attributes and resources in the activity definition) are not presented in descriptions of referencing objects. For clarity purposes, they will be shown in more detail in their own, separate sections.

ActivityDefinition

$$\langle i_1, ActivityDefinition, \{\langle i_2, name, "string"\rangle, \langle i_3, description, "string"\rangle, \langle i_4, work,$$
$$(\ldots SBQLcode\ldots)\rangle, \langle i_5, performer, (\ldots SBQLcode\ldots)\rangle, \langle i_6, attributes, \{i_7\}\rangle,$$
$$\langle i_8, resources, \{i_9\}\rangle\}\rangle$$

name	**type**	**description**
id	integer	internal, not readable identifier of the activity definition
ACTIVITY_DEFINITION	flag	flag indicating that this object is an activity definition
name	string	unique, externally readable and meaningful name of the activity definition
description	string	description of the activity definition
work	SBQL Procedure	the SBQL procedure that will be executed for all instances of this activity definition - in case of an automatically performed activities, after preconditions have been checked and before nested activities of this activity are launched; this element constitutes an atomic work unit in case of automatic activity, but it can consist of many procedure calls nested hierarchically (eg. perform some business logic, call external systems or make operations on workflow data)

(continued)

performer	SBQL query	performer who will perform activities with this definition – may be specified in numerous ways, eg. depending on role, qualifications or other properties; this attribute is used as a definition for eligible performers - actual resource that will be assigned to perform particular task will be determined dynamically at runtime, using this query
attributes	&Attribute [0..*]	collection of attribute definitions that describe particular activity definition; attributes also hold data that is related to activities and can be accessed from within other activities of the process
resources	&ResourceDefinition [0..*]	non-human resources needed to complete particular activity; activity definition can for example contain 2 references to resource definitions - first one would contain information that 100 sheets of paper are needed, and the second one, that 1 conference room is required; at runtime, those resources would be looked up in the available resources' pool and locked for the time of activity execution; after the activity has completed, all resources would be returned to the pool, so that other activities could allocate them

Activity

$$\langle i_1, Activity, \{\langle i_2, name, "string"\rangle, \langle i_3, description, "string"\rangle, \langle i_4, definition, i_{123}\rangle,$$
$$\langle i_5, preconditions, (...SBQLcode...)\rangle, \langle i_6, postconditions, (...SBQLcode...)\rangle,$$
$$\langle i_7, processes, \{i_{223}, i_{224}, i_{230}\}\rangle, \langle i_8, followingActivities, \{i_{400}, i_{401}\}\rangle,$$
$$\langle i_9, precedingActivities, \{i_{398}, i_{397}\}\rangle, \langle i_{10}, performer, i_{686}\rangle,$$
$$\langle i_{11}, resources, \{i_{789}, i_{800}\}\rangle, \langle i_{12}, attributes, \{i_{900}, i_{901}\}\rangle, \langle i_{13}, startDate, 2006{-}12{-}11T11:23:43\rangle,$$
$$\langle i_{14}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{15}, status, (...SBQLcode...)\rangle,$$
$$\langle i_{16}, instances, (...SBQLcode...)\rangle, \langle i_{17}, errors, \{\langle i_{345}, error, "errormessage"\rangle\}\rangle$$

name	**type**	**description**
id	integer	internal, not readable identifier of the activity
ACTIVITY	flag	flag indicating that this object is an activity
name	string	unique, readable and meaningful name of the activity
description	string	description of the activity
definition	&ActivityDefinition	definition of the instantiated activity
preconditions	SBQL query	an SBQL query that returns true, if the conditions have been satisfied, or otherwise false; there can be many SBQL queries (for many preconditions) connected with logical operators; if specified so, workflow management system would have to wait until particular conditions are satisfied (eg. invoice from other company is received)
postconditions	SBQL query	same as above, but checked at the end of activity execution
processes	&ProcessDefinition [0..*]	nested subprocesses that are to be executed within this activity; these processes can contain arbitrary number of theoretically infinitely nested processes that have already been defined, stored in the system, and can be easily plugged into new processes at design time

(continued)

followingActivities	&Activity [0..*]	collection of activities that will be executed after the current one completes; if no conditions are specified in the following activities, they will all be executed simultaneously, if only available resources suffice; simple or more advanced routing constructs can be modeled by using preconditions in each of the following activities
precedingActivities	&Activity [0..*]	collection of activities that have already been executed prior to execution of the current activity; useful when one wants to refer to data contained in attributes of the preceding activities or check those activities' statuses
performer	&Performer	reference to object of class **Performer** – already allocated resource as a result of executing query defined in the **performer** attribute of this activity's definition; external systems using this workflow management system can for example query all tasks for presence of given performer - this will create list of particular employee's activities (the notion of worklist)
resources	&Resource [0..*]	references to resources required and allocated by this activity; resources allocated by one activity, cannot be reallocated by another one, unless they are first deallocated as a consequence of activity (that previously allocated given resource) completion or failure
attributes	&Attribute [0..*]	collection of current activity's attributes together with their values - all the activity-scoped data held and transfered within the workflow processes is kept in this element

(continued)

startDate	datetime	current activity's start time and date
finishDate	datetime	current activity's end time and date
status	SBQL query	execution status of current activity – may have several possible values ("completed", "deferred", "error" or other), or may return one of above values calculated by a query
instances	SBQL query	query that returns number of instances that this particular activity is required to run - may be supplied in the form of a number or for example query that returns value of an activity or process attribute
errors	string [0..*]	collection of errors that have occurred during execution of the current activity; this collection is checked after the process completion, and the appropriate exception handlers (compensation procedures) are executed, if specified

ProcessDefinition

$$\langle i_1, ProcessDefinition, \{\langle i_2, activities, \{i_{81}, i_{82}, i_{83}\}\rangle, \langle i_3, attributes, \{i_{91}, i_{92}, i_{93}\}\rangle,$$
$$\langle i_4, resourceAllocation, "allocation"\rangle, \langle i_5, compensation, (...SBQLcode...)\rangle\}\rangle$$

name	**type**	**description**
id	integer	internal and not readable identifier of the process definition
PROCESS_DEFINITION	flag	flag indicating that this object is a process definition
activities	&Activity [0..*]	activities to be carried out during process execution; all activities from this collection are executed simultaneously; each of them can contain **followingActivities** that will be executed after the first ones are completed
attributes	&Attribute [0..*]	collection of all possible attributes for this particular process definition (without values or with default values)
resourceAllocation	string	method for allocating resources - can be set to several values, for example "load_balance", "case_handling" or "random"; the process manager module would allocate tasks to performers basing on the algorithm specified within this element
compensation	SBQL Procedure	a fragment of code that will be executed upon failure of the process execution

Process

$\langle i_1, Process, \{\langle i_2, name, "string"\rangle, \langle i_3, definition, i_4\rangle, \langle i_5, startDate, 2006{-}12{-}11T11:23:43\rangle,$

$\langle i_6, endDate, 2006-12-11T11:32:11\rangle, \langle i_7, attributes, \{i_8, i_9, i_{10}\}\rangle\}\rangle$

name	**type**	**description**
id	integer	internal and not readable identifier of the process instance
PROCESS	flag	flag indicating that this object is a process instance
name	string	unique name of the process instance
definition	&ProcessDefinition	reference to the definition of this process
startDate	datetime	date and time when the process has started
endDate	datetime	date and time when the process execution ended
attributes	&Attribute [0..*]	collection of process attributes, includes values; this element constitutes process-scoped data that can be accessed from within any activity of this process instance

Attribute

$\langle i_1, Attribute, \{\langle i_2, type, type\rangle, \langle i_3, name, "string"\rangle, \langle i_4, comment, "string"\rangle, \langle i_5, value, i6\rangle\}\rangle$

name	**type**	**description**
id	integer	internal and not readable identifier of the attribute
type	type	type of the attribute variable
name	string	name of the attribute
comment	string	additional comment on the attribute
value	(any type)	value of the attribute

Resource

$\langle i_1, Resource, \{\langle i_2, definition, i_3\rangle, \langle i_4, quantity, 123\rangle\}\rangle$

name	**type**	**description**
id	integer	internal and not readable identifier of the resource
definition	&ResourceDefinition	reference to definition of this resource
quantity	real	quantity of a particular resource that is needed

ResourceDefinition

$$\langle i_1, ResourceDefinition, \{\langle i_2, name, "string"\rangle, \langle i_4, availableQuantity, 523\rangle\}\rangle$$

name	type	description
id	integer	internal and not readable identifier of the resource definition
name	string	reference to definition of this resource
availableQuantity	real	quantity of a particular resource that is currently available

Example process

In chapter 2 we have written an example process in the extended SBQL notation. The code below, written in the Stack-Based Approach notation, illustrates results of executing that mentioned code, as it would be persisted in the data store.

$$\langle i_1, ProcessDefinition, \{\langle i_2, activities, \{i_{57}\}\rangle, \langle i_4, attributes, \{i_{13}, i_{14}\}\rangle, \langle i_6, resourceAllocation, "random"\rangle\}\rangle$$

$$\langle i_7, Process, \{\langle i_8, name, "Insurance\ Claims\ Handling"\rangle, \langle i_9, definition, i_1\rangle, \langle i_{10}, startDate, 2006-12-11T11:23:43\rangle, \langle i_{11}, endDate, 2006-12-14T11:32:11\rangle\}\rangle$$

$$\langle i_{15}, ActivityDefinition, \{\langle i_{16}, name, "Register\ Claim"\rangle, \langle i_{17}, description, "Register\ incoming\ insurance\ claim"\rangle, \langle i_{18}, performer, (\ldots SBQL\ code \ldots)\rangle\}\rangle$$

$$\langle i_{19}, ActivityDefinition, \{\langle i_{20}, name, "Classify\ Claim"\rangle, \langle i_{21}, description, "Classify\ whether\ the\ incoming\ claim\ is\ simple\ or\ complicated"\rangle, \langle i_{22}, performer, (\ldots SBQL\ code \ldots)\rangle\}\rangle$$

$$\langle i_{23}, ActivityDefinition, \{\langle i_{24}, name, "Phone\ Garage"\rangle, \langle i_{25}, description, "Phone\ garage\ in\ order\ to\ obtain\ information\ about\ damages"\rangle, \langle i_{26}, performer, (\ldots SBQL\ code \ldots)\rangle, \langle i_{27}, resources, \{i_{28}\}\rangle\}\rangle$$

$$\langle i_{29}, ActivityDefinition, \{\langle i_{30}, name, "Check\ Insurance"\rangle, \langle i_{31}, description, "Check\ vehicle's\ insurance\ policy"\rangle, \langle i_{32}, work, (\ldots SBQL\ code \ldots)\rangle, \langle i_{33}, performer, "AUTO"\rangle, \langle i_{34}, attributes, \{i_{35}\}\rangle\}\rangle$$

$\langle i_{36}, ActivityDefinition, \{\langle i_{37}, name, "Check\ History"\rangle, \langle i_{38}, description, "Check\ vehicle's\ history"\rangle, \langle i_{39}, performer, (\ldots SBQL\ code\ldots)\rangle, \langle i_{40}, attributes, \{i_{41}\}\rangle\}\rangle$

$\langle i_{42}, ActivityDefinition, \{\langle i_{43}, name, "Decide"\rangle, \langle i_{44}, description, "Make\ decision\ whether\ the\ claim\ should\ be\ accepted\ or\ rejected"\rangle, \langle i_{45}, performer, (\ldots SBQL\ code\ldots)\rangle\}\rangle$

$\langle i_{46}, ActivityDefinition, \{\langle i_{47}, name, "Pay"\rangle, \langle i_{48}, description, "Pay\ money\ to\ claim\ originator"\rangle, \langle i_{49}, performer, (\ldots SBQL\ code\ldots)\rangle\}\rangle$

$\langle i_{50}, ActivityDefinition, \{\langle i_{51}, name, "Send\ Letter"\rangle, \langle i_{52}, description, "Send\ letter\ to\ the\ claim\ originator"\rangle, \langle i_{53}, performer, (\ldots SBQL\ code\ldots)\rangle, \langle i_{54}, resources, \{i_{55}, i_{56}\}\rangle\}\rangle$

$\langle i_{57}, Activity, \{\langle i_{58}, name, "Register\ Claim"\rangle, \langle i_{59}, definition, i_{15}\rangle,$
$\langle i_{60}, preconditions, (...SBQL\ code...)\rangle, \langle i_{61}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{62}, followingActivities, \{i_{68}\}\rangle, \langle i_{64}, performer, i_{300}\rangle,$
$\langle i_{65}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{66}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{67}, status, "NONE"\rangle\}$

$\langle i_{68}, Activity, \{\langle i_{69}, name, "Classify\ Claim"\rangle, \langle i_{70}, definition, i_{19}\rangle,$
$\langle i_{71}, preconditions, (...SBQL\ code...)\rangle, \langle i_{72}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{73}, followingActivities, \{i_{78}, i_{90}\}\rangle, \langle i_{74}, precedingActivities, \{i_{57}\}\rangle,$
$\langle i_{75}, performer, i_{300}\rangle, \langle i_{76}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{77}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{78}, status, "NONE"\rangle,$

$\langle i_{78}, Activity, \{\langle i_{79}, name, "Phone\ Garage"\rangle, \langle i_{80}, definition, i_{23}\rangle,$
$\langle i_{81}, preconditions, (...SBQL\ code...)\rangle, \langle i_{82}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{83}, followingActivities, \{i_{114}\}\rangle, \langle i_{84}, precedingActivities, \{i_{68}\}\rangle, \langle i_{85}, performer, i_{301}\rangle,$
$\langle i_{86}, resources, \{i_{365}\}\rangle, \langle i_{87}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{88}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{89}, status, "NONE"\rangle\}\rangle$

$\langle i_{90}, Activity, \{\langle i_{91}, name, "Check\ Insurance"\rangle, \langle i_{92}, definition, i_{29}\rangle,$
$\langle i_{93}, preconditions, (...SBQL\ code...)\rangle, \langle i_{94}, postconditions, (...SBQL\ code...)\rangle,$

$\langle i_{95}, followingActivities, \{i_{102}, i_{114}\}\rangle, \langle i_{96}, precedingActivities, \{i_{68}\}\rangle, \langle i_{97}, performer, "AUTO"\rangle,$
$\langle i_{98}, attributes, \{i_{35}\}\rangle, \langle i_{99}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{100}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{101}, status, "NONE"\rangle\}\rangle$

$\langle i_{102}, Activity, \{\langle i_{103}, name, "Check\ History"\rangle, \langle i_{104}, definition, i_{36}\rangle,$
$\langle i_{105}, preconditions, (...SBQL\ code...)\rangle, \langle i_{106}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{107}, followingActivities, \{i_{78}\}\rangle, \langle i_{108}, precedingActivities, \{i_{90}\}\rangle, \langle i_{109}, performer, i_{301}\rangle,$
$\langle i_{110}, attributes, \{i_{41}\}\rangle, \langle i_{111}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{112}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{113}, status, "NONE"\rangle\}\rangle$

$\langle i_{114}, Activity, \{\langle i_{115}, name, "Decide"\rangle, \langle i_{116}, definition, i_{42}\rangle,$
$\langle i_{117}, preconditions, (...SBQL\ code...)\rangle, \langle i_{118}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{119}, followingActivities, \{i_{125}, i_{136}\}\rangle, \langle i_{120}, precedingActivities, \{i_{78}, i_{90}\}\rangle, \langle i_{121}, performer, i_{302}\rangle,$
$\langle i_{122}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{123}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{124}, status, "NONE"\rangle\}\rangle$

$\langle i_{125}, Activity, \{\langle i_{126}, name, "Pay"\rangle, \langle i_{127}, definition, i_{46}\rangle,$
$\langle i_{128}, preconditions, (...SBQL\ code...)\rangle, \langle i_{129}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{130}, followingActivities, \{i_{136}\}\rangle, \langle i_{131}, precedingActivities, \{i_{114}\}\rangle, \langle i_{132}, performer, i_{301}\rangle,$
$\langle i_{133}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{134}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{135}, status, "NONE"\rangle\}\rangle$

$\langle i_{136}, Activity, \{\langle i_{137}, name, "Send\ Letter"\rangle, \langle i_{138}, definition, i_{50}\rangle,$
$\langle i_{139}, preconditions, (...SBQL\ code...)\rangle, \langle i_{140}, postconditions, (...SBQL\ code...)\rangle,$
$\langle i_{141}, followingActivities, \{\}\rangle, \langle i_{142}, precedingActivities, \{i_{114}, i_{125}\}\rangle, \langle i_{143}, performer, i_{301}\rangle,$
$\langle i_{144}, resources, \{i_{401}, i_{402}\}\rangle, \langle i_{145}, startDate, 2006-12-11T11:23:43\rangle,$
$\langle i_{146}, finishDate, 2006-12-11T11:32:11\rangle, \langle i_{147}, status, "NONE"\rangle\}\rangle$

$\langle i_{13}, Attribute, \{\langle i_{148}, name, "classification"\rangle, \langle i_{149}, type, string\rangle,$
$\langle i_{150}, comment, "Classification\ whether\ the\ claim\ is\ simple\ or\ complex"\rangle\}$

$\langle i_{14}, Attribute, \{\langle i_{151}, name, "decision"\rangle, \langle i_{152}, type, boolean\rangle,$
$\langle i_{153}, comment, "Decision\ whether\ the\ claim\ should\ be\ accepted\ or\ rejected"\rangle\}$

$\langle i_{35}, Attribute, \{\langle i_{154}, name, "policiesFound"\rangle, \langle i_{155}, type, boolean\rangle,$
$\langle i_{156}, comment, "Vehicle\ insurance\ policies\ that\ have\ been\ found\ in\ the\ database"\rangle\}$

$\langle i_{41}, Attribute, \{\langle i_{157}, name, "vehicleHistory"\rangle, \langle i_{157}, type, string\rangle,$
$\langle i_{158}, comment, "Insurance\ history\ of\ the\ vehicle"\rangle\}$

A.2 Stack-Based Query Language Extensions

The proposed Stack-Based Query Language extension, as described in previous chapters, involves numerous new language constructs for creating and manipulating workflow processes and activities. All of them are described in this section, by using recursive descent in the EBNF notation. Two of the non-terminals, `query` and `identifier`, have not been further described, as they are self-explanatory and easy to understand. The `query` denotes any valid SBQL code, whereas `identifier` means any legal SBQL identifier.

```
create_activity_definition ::= "{" name [description] [work] performer [attributes]
                               [subprocesses] [resources] "}" "as" identifier ;
create_activity            ::= "create activity" "{" name [description] definition
                               [preconditions] [postconditions]
                               [preceding_activities] [following_activities]
                               [instances] "}" "as" identifier ;
create_process_definition  ::= "create process definition" "{" name [allocation]
                               activities [attributes] [compensation] "}" "as"
                               identifier ;
create_process_instance    ::= "create process" "{" name definition "}" "as"
                               identifier ;
launch_process             ::= "launch" " " identifier ;
cancel_process             ::= "cancel" " " identifier ;

name                       ::= "name" " " string ;
description                ::= "description" " " string ;
performer                  ::= "performer" "{" query "}" ;
attributes                 ::= "attributes" "{" {identifier : type} ";" "}" ;
subprocesses               ::= "subprocesses" "{" "bag" "{" {identifier} "}" "}" ;
resources                  ::= "resources" "{" query "}" ;
definition                 ::= "definition" identifier ;
preconditions              ::= "preconditions" "{" condition "}" ;
condition                  ::= query | "waitfor" "{" query "}" [query] ;
postconditions             ::= "postconditions" "{" condition "}" ;
activities                 ::= "activities" "{" "bag" "{" {identifier} "}" "}" ;
preceding_activities       ::= "preceding_activities" "{" activities "}" ;
following_activities       ::= "following_activities" "{" [activities] "}" ;
instances                  ::= "instances" "{" query "}" ;
allocation                 ::= "allocation" " " string ;
compensation               ::= "compensation" "{" query "}" ;
```

Additionally, there are several reserved words that can be used when querying the database for information related to workflow processes. Naming convention which they represent is consistent with the entity names, described earlier in this appendix. The words are listed below.

```
Process
ProcessDefinition
Activity
ActivityDefinition
Resource
ResourceDefinition
```

B Workflow process schema documentation

process-def	Complex type	Process definition specification
id	xs:string	Unique identifier of the process definition
activities	activity[1..*]	Activities that constitute this process
attributes	attribute-def[0..*]	Definition of process-scoped attributes
resource-allocation	xs:string	Specification of resources allocation algorithm
exception-handlers	exception-handler	Procedures defined declaratively that handle exceptional situations
process	Complex type	Process instance specification
id	xs:string	Unique identifier of the process instance
definition	process-def	Definition, according to which process should be instantiated
name	xs:string	Readable and meaningful name of the process instance
start-date	xs:dateTime	Date and time when the process execution started
end-date	xs:dateTime	Date and time when the process execution ended
attributes	attribute[0..*]	Process scoped attributes with their values
activity-def	Complex type	Activity definition specification
id	xs:string	Unique identifier of the activity definition
name	xs:string	Readable and meaningful name of the activity definition
description	xs:string	Additional description of the activity definition written in natural language
class	xs:string	Java class that will be instantiated and run upon execution of activities defined according to this definition
performer	performer	Specification of performer allocation rules that will be evaluated during process execution, and used to allocate relevant performer to an activity
attributes	attribute-def[0..*]	Definition of activity-scoped attributes
resources	resource	Specification of resources required to carry out an activity; these allocation rules will be evaluated during process execution, and appropriate resources will be allocated to an activity

activity	Complex type	Activity instance specification
id	xs:string	Unique identifier of the activity instance
name	xs:string	Readable and meaningful name of the activity instance
description	xs:string	Additional description of the activity definition written in natural language
definition	activity-def	Definition, according to which activity should be instantiated
preconditions	condition[0..*]	Specification of conditions that will be evaluated before activity execution - when successful, activity guarded by this condition will be attained
postconditions	condition[0..*]	Specification of conditions that will be evaluated after activity execution - when successful, process execution will be able to proceed to another activities
nested-processes	xs:string[0..*]	Enables inclusion of subprocesses that will be executed within this activity
following-activities	activity[0..*]	Specifies which activities will have to be enacted after the current one completes
preceding-activities	xs:string[0..*]	Specifies activities that are immediately preceding to the current one. To avoid cyclic references, this element has been simplified to contain only identifiers of activity instances
performer	xs:string	Contains identifier of specific performer, evaluated by the resource management module, according to performer specification contained in the activity definition
resources	resource[0..*]	Contains identifiers of allocated resources together with each resource's needed quantity; calculated during process execution by the resource management module
attributes	attribute[0..*]	Activity scoped attributes with their values
start-date	xs:dateTime	Date and time when the activity execution started
finish-date	xs:dateTime	Date and time when the activity execution finished
status	xs:string	Status of the current activity, set by the process management module (eg. "COMPLETED", "WAITING", etc.)
errors	error[0..*]	Sequence of errors that were encountered during execution of current activity
instances	instances	Specification of the number (or method to evaluate the number) of instances that this activity is required to run

resource	Complex type	Resource specification
id	xs:string	Unique identifier of the resource
name	xs:string	Readable and meaningful name of the resource
quantity	xs:double	Amount of the resource that is available or required (depending on context)
performer	Complex type	Specifies performer allocation method (only one from the list below)
roles	xs:string[1..*]	Specifies list of roles, whose users are eligible for allocation to particular task
direct	xs:string	Specifies identifier of single performer, who will receive the task
capabilities	anonymous type[1..*]	Defines capabilities, according to which the performer will be selected; each capability consists of elements `capability-name`, `capability-value`, and `capability-relation`, each of type xs:string
attribute	xs:string	Specifies attribute that would contain name of the particular performer
condition	Complex type	Specifies condition according to time events, attribute values, activity statuses, or basing on results of custom Java code execution; contains attribute `wait` of type xs:boolean, which specifies if the workflow engine needs to wait for this condition to be satisfied
time	anonymous type	Condition based on time events; handles two kinds of events: `delay` (`xs:positiveInteger`) or `absolute` (`xs:dateTime`)
attribute	anonymous type	Condition based on attribute's value; contains the following subelements: `name` (for attribute's name), `value` (value to be compared with actual attribute's value), and `relation` (to specify relation between actual and expected attribute's value)
status	anonymous type	Condition based on activity's status; contains two subelements: `activity` (denotes which activity's status must be checked), and `value` (specifies what is the expected attribute's status)
completed-preceding	xs:integer	Condition based on the number of completed preceding activities
code	xs:string	Custom condition check, written in Java and supplied as a class name implementing the `CodeCondition` interface

or	anonymous type	Collection of conditions joint together with logical OR operation
and	anonymous type	Collection of conditions joint together with logical AND operation
xor	anonymous type	Collection of conditions joint together with logical XOR (exclusive or) operation
available-resources	Complex type	Specification of resources that are available for allocation
users	user[0..*]	Collection of performers that are available for task allocation
resources	resource[0..*]	Collection of resources that are available to be allocated to activities
roles	role[0..*]	Collection of roles that are present in the system and may contain users assigned to them
role	Complex type	Represents role, can have many users assigned
id	xs:string	Unique identifier of the role
name	xs:string	Meaningful and readable name of the role
user	Complex type	Represents user
id	xs:string	Unique identifier of the user
name	xs:string	User's name
surname	xs:string	User's surname
email	xs:string	User's email address
address	xs:string	User's mailing address
capabilities	anonymous type[0..*]	Collection of user's capabilities, each of which consists of two elements: `capability-name`, and `capability-value`
roles	role	Collection of roles, to which the user belongs
attribute	Complex type	Specifies data container together with its value visible in particular scope
definition	attribute-def	Definition of this attribute
value	xs:string	Value of this attribute
attribute-def	Complex type	Specifies definition of an attribute, common to all attributes that reference it
name	xs:string	Defines name for an attribute
class	xs:string	Specifies Java class name for the attribute's value
comment	xs:string	Additional comment, describing responsibility of attributes sharing this definition

exception-handler	Complex type	Defines handling rules for exceptions that occurred during process execution
exception	xs:string	Java class name representing type of the encountered exception; usually subclass of java.lang.Throwable
handler	xs:string	Java handler class, which will be launched upon execution of relevant compensation procedure
error	Complex type	Specification of a problem that was encountered during process execution
class	xs:string	Denotes class of the exception that has been thrown during process execution
message	xs:string	Message contained within the thrown exception object
stacktrace	xs:string	Stack trace of the thrown exception; for logging or debugging purposes
instances	Complex type	Specifies number of instances for particular activity to be executed; may contain only one element: `attribute` or `number`
attribute	xs:string	Specifies which attribute contains number of instances for particular activity to execute (evaluation deferred until runtime)
number	xs:integer	Contains a specific number of instances for an activity to run (specified at design time)

C API documentation for the process and resource management modules

C.1 Process Manager

- `public void createProcessInstance(Process p)` - Creates a new process instance in the data store according to supplied `Process` object. Such instance may be launched and further handled by the process management engine.

- `public void launchProcess(Process p)` - Launches an already created process instance, which is provided as a parameter.

- `public void launchProcess(String processId)` - Launches an existing process instance. Unlike the previous method, it requires only the process identifier to be specified.

- `public void launchActivity(Activity activity, Process enclosingProcess)` - Starts execution of specified activity within the specified process instance. This method, unlike the preceding two, is not intended to be externally invoked, and thus is not exposed in the component's remote interface.

- `public void modifyActivity(Activity a, Process enclosingProcess)` - Updates an existing activity with data contained in the specified activity object, and which exists inside the specified process instance.

- `public void completeActivity(Activity act, Process enclosingProcess)` - Sets activity's status to "COMPLETED", so that the process could proceed with execution. Useful when task is not automatically carried out by the system, but requires some human actions to be taken (eg. sending letter by mail).

- `public Activity findActivity(String id, Process enclosingProcess)` - Searches for an activity within specified process instance, and returns single activity object.

- `public String getProcessAttribute(String name, Process process)` - Returns value of process-scoped variable by the specified attribute name.

- `public void setProcessAttribute(String name, String value, Process process)` - Sets value of process-scoped attribute, according to specified process instance and variable name.

- `public String getActivityAttribute(String name, Activity activity)` - Returns value of activity-scoped variable by the specified attribute name.

- `public void setActivityAttribute(String name, String value, Activity activity)` - Sets value of activity-scoped attribute, according to specified activity instance and variable name.

- `public void createActivityAttribute(String name, String type, String comment, Activity activity)` - Creates attribute in the activity scope, so that it can be written and read later during process execution. Such specification must include name, type and comment of the attribute for integrity checking purpose.

- `public void createProcessAttribute(String name, String type, String comment, Process process)` - Creates attribute in the process scope, so that it can be written and read later during process execution. Such specification must include name, type and comment of the attribute for integrity checking purpose.
- `public Process getProcess(String processId)` - Returns process instance object by the supplied unique process identifier.

C.2 Resource Manager

- `public void allocateResources(Activity activity, Process enclosingProcess) throws InsufficientResourcesException` - Allocates resources required to attain particular task to the specified activity object within supplied process instance. This method is used internally by the process management module, and hence should not be invoked externally. It is not exposed in the component's remote interface.
- `public void allocatePerformer(Process process, Activity activity, String allocation)` throws InsufficientResourcesException - Allocates performer required to attain particular task to the specified activity object within supplied process instance, and in accordance to specified allocation algorithm (random, round robin, load balance, etc.). This method is used internally by the process management module, and hence should not be invoked externally. It is not exposed in the component's remote interface.
- `public void createUser(User user)` - Creates user object in the underlying data store. Such created user can be allocated to tasks as a performer.
- `public void removeUser(String login)` - Finds user by his or her login and removes them from the data store.
- `public void updateUser(User user)` - Modifies existing user information with those contained in the supplied object.
- `public User getUser(String login)` - Retrieves single user object from the data store and returns it to the caller.
- `public Collection<User> getAllUsers()` - Retrieves all users that exist in the data store and returns them in the form of collection.
- `public void createRole(Role role)` - Creates role object in the underlying data store. Users existing in the system can then be added to such created role.
- `public void removeRole(String roleName)` - Finds role by its name and removes it from the data store.
- `public void updateRole(Role role)` - Modifies existing role information with those contained in the supplied role object.
- `public Role getRole(String roleName)` - Retrieves single role object from the data store and returns it to the caller.
- `public Collection<Role> getAllRoles()` - Retrieves all roles that exist in the data store and returns them in the form of collection.

- `public void createResource(Resource resource)` - Creates resource object in the underlying data store. Created resources can then be allocated to activities.
- `public void removeResource(String resourceName)` - Finds resource by its name and removes it from the data store.
- `public void updateResource(Resource resource)` - Modifies existing resource information with those contained in the supplied role object.
- `public Resource getResource(String resourceName)` - Retrieves single resource object from the data store and returns it to the caller.
- `public Collection<Resource> getAllResources()` - Retrieves all resources that exist in the data store and returns them in the form of collection.
- `public void deallocateResources(Activity activity)` - Deallocates all resources from the specified activity, so that they become eligible for reuse. This method, like some of the previously described ones, is not intended for external use, but is called from within the process management module. Also, it is not exposed in the component's remote interface.