KDPMEL: A Knowledge Discovery Process Modeling and Enacting Language

Hesham A. Mansour
FJA-US, Inc., 1040 Avenue of the Americas, 4th Floor, New York, NY 10018, USA

Abstract - *Knowledge Discovery in Databases (KDD) is a complex process that is highly interactive and iterative. Managing the process and its underlying resources and interacting activities is a very complex task. Today, KDD projects are typically approached in an unstructured, ad hoc manner due to a lack of formal methods and methodologies for their development. The similarities between KDD processes and software development processes suggest that approaches used to manage the development of software processes, such as process programming and process-centered support environments, are also applicable and in fact advantageous to KDD processes. This paper proposes the Knowledge Discovery Process Modeling and Enacting Language (KDPMEL) that is designed specifically to provide both the syntax and semantics needed to capture and execute KDD processes. KDPMEL combines aspects of both process programming and KDD to represent KDD processes precisely as process programs. Along with KDPMEL, an Integrated Development Environment (IDE) for KDPMEL is proposed to enable and facilitate the development and execution of KDPMEL programs. We illustrate and evaluate KDPMEL by representing and executing illustrative examples of KDD activities and tasks.*

Keywords: Data Mining, Knowledge Discovery in Databases (KDD), KDD Process, Process Programming

1 Introduction

The complexities of KDD processes have been recognized and as a result, various approaches and methodologies have been proposed [1] to provide user guidance and support for the development of KDD processes.

Unfortunately, while many KDD support systems have been proposed and prototyped, the support they provide either targets individual activities performed within the process---*activity-oriented* support---or is based on architectures that support a predefined generic process model---*KDD support environments*. There have been a few proposals that apply *process-oriented* support to KDD. Among these, only [3] uses a process language approach based on Little-JIL to explicitly represent the coordination aspect of KDD processes. Each of these approaches has certain limitations and lacks either the proper formalism and/or the suitable semantics needed to explicitly represent and effectively support the performance of KDD processes.

In activity-oriented support, the process concept is neither enforced nor endorsed and is only represented in documentation and guidelines. In contrast, while the process concept is adopted by KDD support environments, it is limited to a generic form that is described by the major phases and tasks of a designated process model. In addition, these environments usually support particular KDD techniques and have a prescribed set of supporting tools that are typically built in the environment or tightly integrated with it. Although the process-oriented support presented in [3] is very promising, it concentrates on supporting only the coordination aspect of the process. In addition to the discovered deficiencies in Little-JIL, only the simplest processes can be modeled visually using Little-JIL. For additional information about the different approaches for supporting KDD processes and their limitations, see [1].

The similarities between KDD processes and software processes suggest that approaches used to manage and support the development of software processes are also applicable and in fact advantageous to KDD processes. Although some researchers [3], [6]-[8] have recognized these similarities, none (to our knowledge) has proposed a comprehensive approach for supporting KDD processes through a dedicated KDD process programming language that can combine aspects from both process technology and KDD to explicitly represent and support KDD processes more accurately. This language represents a core component of a much larger process-centered support environment that can enable and facilitate the modeling, management, and enactment of KDD process programs written in this language.

This paper proposes the Knowledge Discovery Process Modeling and Enacting Language (KDPMEL) that is designed specifically to provide both the syntax and semantics needed to capture and execute KDD processes. KDPMEL provides predefined language constructs for modeling the resources of a process, encoding its steps and specifying their sequencing, invoking external tools, and specifying the usage of resources by the steps of the process. KDPMEL combines aspects of both process programming and KDD to represent KDD processes precisely as process programs. KDPMEL supports modeling KDD tasks at different levels of detail and abstraction in order to specify both generic and specialized tasks. KDPMEL provides special control constructs to explicitly model task dependencies on other tasks. This feature is particularly important for KDD and is intended to effectively manage the dependencies between KDD techniques. Having these dependencies explicitly defined can assure that they are appropriately handled. We believe that the above mentioned features precisely suit KDD and their existence in a process language that combines process aspects along with KDD aspects is novel for implementing KDD processes.

Our philosophy in designing a process language for KDD instead of utilizing or adapting an existing process language is that we want the language to be highly expressive for the essential aspects of KDD and not just the process-aspect. The paper is structured as follows. This section provides background information and the motivation for our work. Section 2 presents KDPMEL and illustrates its usage in modeling KDD processes. Section 3 outlines an Integrated Development Environment (IDE) for KDPMEL. Section 4 briefly evaluates KDPMEL and provides some lessons earned during its implementation. Section 5 concludes the paper.

2 The Knowledge Discovery Process Modeling and Enacting Language (KDPMEL)

KDPMEL is a novel process programming language for modeling and enacting KDD activities along with their resources, interactions, coordination, and dependencies. The process aspects of the language such as process structuring, task ordering, and artifact management are similar to those found in general process languages, such as Little-JIL [3] and PML [4]. The KDD aspects of the language are specific features for modeling KDD artifacts, tools, and tasks.

KDPMEL provides multiple granularity levels for process decomposition that is particularly suited to KDD. The first level allows for organizing the process into a number of phases (Lifecycle level). The second level allows for defining the generic tasks of each phase (Generic Task level). Finally, the third level allows for defining the specialized tasks for a generic task (Specialized Task level).

A process program written in KDPMEL is organized into three major sections that specify the resources (*artifacts*, *roles*/*actors*, and *tools*) of the process, general information about the process (*goal*, *input*, *outcome*, and *assessment*), and the steps (*activity*, *action*, and *command*) of the process along with their sequencing (*sequence*, *parallel*, *choice*, and *loop*) and dependencies (*disallow*, *require*, and *enable*).

2.1 Language Goals

The major goals of the KDPMEL are the *simplicity*, *flexibility*, *expressiveness*, and *generality*.

2.2 Language Approach

KDPMEL combines both the graphical and process language approaches. Modeling with KDPMEL employs a number of graphical modeling editors on top of a textual process programming language designed specifically for KDD. The goal of the modeling editors is to facilitate the construction and presentation of certain components of the process, as represented by four types of graphs that display the overall process (Process Graph), the resources of the process (Resources Graph), a graph for each activity (Activity Graph), and a graph for each action (Action Graph). Furthermore, a read-only graph that shows the progress of the artifacts within the process (Artifact Flow Graph) is provided. The Process Graph indicates process information such as goal, input, outcome, etc. The Resources Graph shows the artifacts, roles/actors, and tools of the process. Each Activity Graph shows the activity's constituent actions while each Action Graph provides information such as tools utilized by the action, the artifacts consumed and produced by the action, and the actor assigned to the action. In addition to the graphical editors, a number of form-based editors exist (Process, Activity, Artifact, Role/Actor, and Tool Forms) to present and update certain information that is best shown and updated in a form-based style.

Combining different types of editors and views in source-based, graph-based, and form-based styles is novel and allows both technical and non-technical users to participate in the modeling phase of the process. Fig. 1 illustrates the process modeling approach adopted by KDPMEL.

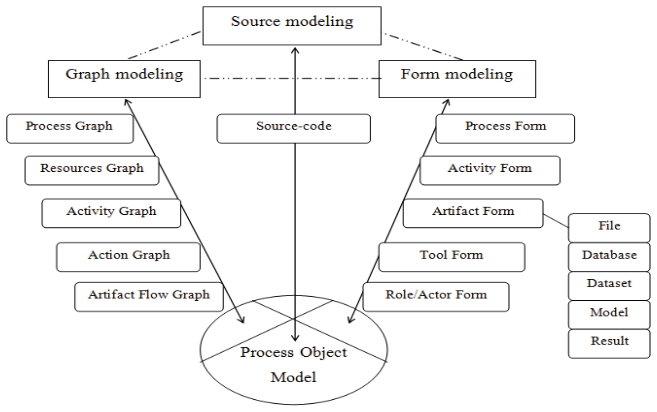

Fig. 1. KDPMEL Process Modeling Approach

As illustrated in Fig. 1, the process object model (the internal object-oriented representation of the process model) is described using source-based modeling, where all the process details can be specified using the KDPMEL source-code editor. The graph-based and form-based modeling approaches are used for certain process parts that are best presented (e.g., overall process structure and control flow) and updated (e.g., artifacts description) using these approaches. Alternatives exist. For instance, the process object model can be created and updated by either the source-based or graph-based approaches, whereas the form-based approach can only update the process object model. The process object model is translated into its source-based, graph-based, and form-based representations through a number of model visitors, which are based on the object-oriented visitor design pattern [15], that navigate the model and perform the appropriate translation. The source-code model visitor performs complete translation for the model to its source-code representation. The graph model visitors create graphs for the process, resources, each activity, each action, and a read-only graph for the flow of artifacts in the process. The form model visitors create forms for the process and for each activity, action, artifact (File, Database, Dataset, Model, and Result), role/actor, and tool. In the figure above, the dashed lines between the different modeling approaches indicate the translations between these approaches.

We believe that this hybrid modeling approach combines the benefits of the underlying approaches and enables specification of high-level process models as well as more complex ones in a manner that is convenient for both technical and non-technical users.

A KDPMEL process program can be developed from scratch using either the source-based or graph-based modeling approach and can be updated using any approach. Any update to the process program in any representation is reflected to the other representations.

The adopted graphical modeling approach tries to avoid ambiguities and mistakes. It also tries to ease the graphical modeling development. Graphical components that are new are inserted into a graph by a drag and drop approach. Graphical components that are already defined but need to be referenced in a graph are inserted into the graph by selecting from a menu that displays appropriate graphical components for the current context. For example, to insert a new actor resource component into the Resources Graph Editor, this component is dragged from a palette and dropped into the editor as shown in Fig. 2.

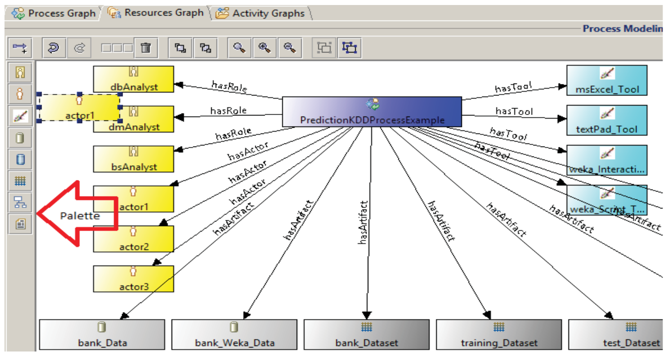

Fig. 2. Inserting an Actor Resource component in The Resources Graph Editor

An example for inserting an actor performer component into the Action Graph Editor is shown in Fig. 3, where a context menu is used to select the type of component to be inserted and another menu is used to select the value of that component.

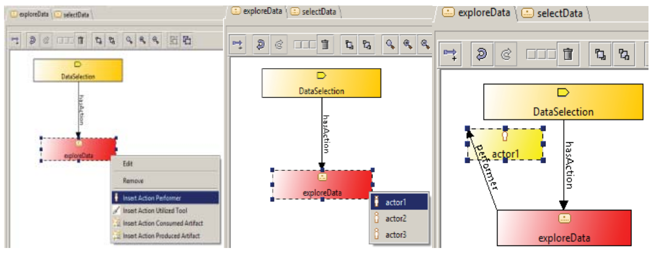

Fig. 3. Inserting an Actor Performer component in the Action Graph Editor

This approach avoids modeling mistakes such as inserting an actor performer component for an actor that was never defined in the resources, if a drag and drop approach were to be used. Even if that actor was already defined in the resources, mistakes can still happen such as specifying the wrong actor name, which was different than what was defined. Moreover, it avoids ambiguities in specifying, for instance, which artifact is consumed and which is produced by the action. If a drag and drop approach were to be used, this case would require the modeler to distinguish between the action's consumed artifact and the other produced one by specifying the correct association name for each artifact, which is more work and can lead to mistakes. Designating a separate association type for each artifact is overwhelming and complicates the graphical modeling.

2.3 KDPMEL Meta-Model (Abstract Syntax)

KDPMEL is defined in terms of a meta-model [11] based on the OMG's SPEM [16] and CWM [10] meta-models, which represents the abstract syntax and static semantics of the language. The abstract syntax is represented by the Unified Modeling Language (UML) [9] class diagrams showing the language metaclasses and their relationships.

2.4 KDPMEL Concrete Syntax

The concrete syntax of KDPMEL is provided in two flavors, textual and graphical, to serve different purposes. The textual concrete syntax (grammar) is useful when specific complex details must be specified. The graphical concrete syntax is easy to understand and use, and is useful for communicating structural and higher level views of the process models created by the language. Also, a form-based interface is provided for process components to allow for presenting and updating the properties of these components.

2.4.1 KDPMEL Textual Concrete Syntax (Grammar)

KDPMEL meta-model provides process specific entities such as *Process*, *Activity*, and *Action*. The *Process* entity represents the entire process and is intended to be used as a container for all the process components. The *Activity* entity represents a composite activity in the process that can be further decomposed into smaller units. The *Action* entity represents a primitive task in the process. The syntax for defining these constructs is given by the following rules:

<process> ::= "**process**" <IDENTIFIER> "{"…"}"

<activity>   ::= “**activity**” <IDENTIFIER> “{“…”}”

<action>   ::= “**action**” <IDENTIFIER> “{“…”}”

For additional information about KDPMEL syntax, see [1].

2.4.2 Modeling KDD Task Dependencies

A KDD task may have dependencies with other tasks in the process. It may *require*, *disallow*, or *enable* other tasks.

The experiment conducted by Jensen et al. [3] on the subject of coordinating interdependent KDD tasks using Little-JIL showed that even a state-of-the-art process language such as Little-JIL lacks the semantics to describe some simple cases easily. Little-JIL lacks semantics for establishing dependencies among tasks and consequently a Little-JIL program must be written in a manner that conforms to these dependencies, which are represented in the structure of the program. A major drawback for this approach is that any change in the process specifications would need substantial modifications for the process program. The authors have acknowledged that adding to Little-JIL process specifications is not as simple as it was hoped to be, at least in the domain of KDD, and requires substantial revisions in process specifications due in part to the interdependence of KDD techniques and their effects. They also stated that changes to the Little-JIL language may be essential to overcome this deficiency.

The KDPMEL approach for explicitly modeling task dependencies is not only more flexible in accommodating changes to the process specifications, but it is more expressive in representing process specifications. The following KDPMEL example specifies a *disallow* dependency between the Three Group Regression (TGR) technique and the Parametric Significance Test:

```
activity LinearRegression {
  choice buildRegressionModel {
    action constructLSRModel {…}
    action constructTGRModel {…
      dependency {
        require …; enable …;
        disallow parametricSignificanceTest;
      }
    }
  }
  choice testRegressionModel {
    action parametricSignificanceTest {…}
    action randomizationTest {…}
  }
}
```

2.4.3 Modeling Specialized KDD Tasks

KDPMEL represents specialized KDD tasks through external commands that are modeled in the program and can be validated and executed through a plug-in mechanism for the tools of these commands.

The following is an example of a KDPMEL action that uses a command to build a decision tree model with the C4.5 (J48) classifier algorithm [12] of the WEKA data mining framework [13]:

```
action buildDecisionTreeModel {
  consume sampleDataset;
  produce sampleDecisionTreeModel;
  performer dmAnalyst;
  utilize {
    call wekaTool {
      command build_C45_Weka_Classifier {
        kind Modeling;
        input sampleDataset;
        output sampleDecisionTreeModel;
        operation"weka.classifiers.trees.J48";
        parameters "-C 0.25 -M 2";
      }
    }
  }
}
```

2.4.4 Modeling Consistency Rules and Constraints for KDD Tasks

The ability to explicitly model and enforce requirements and acceptance criteria of KDD tasks is important for making the KDD process more understandable, evaluating its correctness, assuring its consistent execution, and validating its results.

Requirements and acceptance criteria can be defined for KDPMEL tasks using *pre-conditions* and *post-conditions* constructs to represent constraints that guard entry (task requirements) into and exit (task acceptance criteria) from a task. These constraints are defined by logical expressions over the process artifacts to check their attributes and states.

Artifact State Expression

The predicate *hasState(artifact, state)* checks if an artifact has a particular state. For example, the expression: *hasState(bank_Data, Prepared)* checks if the *bank_Data* artifact has the *Prepared* state.

Artifact Attribute Expressions

Two types of expressions are used to check the attributes of an artifact. The first checks if an attribute is defined in an artifact using the *isDefined(attributeList)* predicate. For example, the expression: !*isDefined(bank_Dataset.id, bank_Dataset.save_act)* checks if the *id* and *save_act* attributes are not defined in the *bank_Dataset* artifact. The second expression type is an attribute comparison expression that compares the value of an attribute with another or with a literal value. For example, the expression: *bank_training_Dataset.instancesPct* == 0.75 && *bank_test_Dataset.instancesPct* == 0.25 checks if the *bank_training_Dataset* artifact has 75% of the instances and the *bank_test_Dataset* artifact has 25% of the instances.

Consistency Rules

Consistency rules are represented as constraints on KDPMEL tasks in the form of pre-conditions and post-conditions that regulate the work of the tasks over the artifacts. In order to maintain valid state changes for an artifact, there should be a *hasState* predicate for that artifact in both the pre-conditions and post-conditions.

2.5 KDPMEL Semantics

Control Flow and Ordering

KDPMEL provides a variety of constructs for specifying the control flow among activities and actions. By using these control constructs, the process control flow can be described at different levels of detail. The activities within a process and the actions within an activity can be grouped using one of the control constructs *sequence*, *parallel*, *choice*, or *loop*. Additionally, activities may be decomposed into a hierarchy of sub-activities and actions.

Ordering

Both the activities within a process and the actions within an activity can be grouped using *sequence*, *parallel*, *choice*, or *loop*. The default ordering is *sequence*.

Sequence

A *sequence* control construct specifies that the execution of its group of tasks (i.e., activities or actions) is performed in the order that they are written in the process program.

Choice

A *choice* control construct specifies that exactly one task is performed from its group of tasks. The decision for determining which task to execute is mainly handled by checking the requirements of each task in the choice. If the requirements for only one of the tasks in the choice are satisfied, then this task is executed. If there are no requirements or if the requirements of two or more tasks are satisfied, then the actor performing the choice must select which task to execute.

Parallel

A *parallel* control construct specifies that its group of tasks can be performed simultaneously.

Loop

A *loop* control construct specifies that its group of tasks can be performed repeatedly. The decision to stop or continue the loop after completing the last task in the loop is handled by checking the requirements of the first task following the loop. The following task may have explicit preconditions and/or implicit requirements such as needed artifacts.

Hierarchical Task Decomposition

An *activity* construct can be decomposed into smaller units of sub-activities and/or actions. This decomposition creates a tree structure of activities and actions. The execution of these tasks is performed depth-first, where the execution of a node in the tree is performed by executing its children in left to right order.

KDD processes may have highly variable control requirements including both strict and loose control. Strict control, in the forms of hierarchical task decomposition and the sequence control construct (flat task decomposition), is needed to specify that certain tasks must be executed in a given order while loose control, for instance, using the *choice* control construct is needed to specify different execution alternatives among the tasks. Both types of decompositions (Flat and Hierarchical) are used to decompose high-level process activities (e.g., the phases of the CRISP-DM [2] process) into lower-level activities and primitive actions (e.g., the generic tasks of the CRISP-DM process), which is a natural way of organizing the process tasks using a divide-and-conquer approach.

Dependency Control Constructs

The dependency control constructs *disallow*, *require*, and *enable* can be associated with an action construct to indicate its dependency requirements on other tasks. The linear regression example given previously specifies a *disallow* dependency between the construction of the Three Group Regression Model (*constructTGRModel*) and the Parametric Significance Test (*parametricSignificanceTest*) technique. The states of KDPMEL tasks and their dependency requirements are recorded during the execution of the tasks. For example, when completing the execution of the *constructTGRModel* action, the following information is recorded for the action:

Action: *constructTGRModel* State: *Completed*

Dependency {Disallow: *parametricSignificanceTest*}

Upon beginning the execution of a task, a test is performed against the completed actions to check whether their *disallow* dependencies prohibit execution of the task. In the example above, when *parametricSignificanceTest* begins this conflict is found, the execution of the action is halted, and control proceeds to the next available action (*randomizationTest*). The *enable* dependency is checked only for the *choice* control construct to determine if one of the choices has been enabled by a completed action. If that was the case, the actor making the choice will be notified. Another test is performed upon beginning the execution of an action to check its requirement dependency (the *require* construct) against the completed tasks to determine if the action can be executed. An action can only be executed if its required tasks are completed successfully.

We believe that this is a novel approach for managing KDD task dependencies that are dynamically reflected at runtime, as opposed to statically structuring these tasks according to their dependencies at modeling time [3], which provides more flexibility not only in modeling time but in execution time also. In addition, it also leads to much shorter programs.

Action Specialized KDD Tasks

KDPMEL has a mechanism to model and execute environment-specific KDD tasks through the use of external commands that can be associated with an *action* and a *tool*. Each tool that is associated with an external command is represented by a plug-in that is invoked to execute the command in the surrounding environment. In the decision tree example given previously, the *build_C45_Weka_Classifier* command is executed by sending the command to the Weka plug-in to perform the execution of the command. This Weka plug-in translates the command into a valid Weka command and executes it. Fig. 4 illustrates this plug-in approach for external tool (Weka) commands.

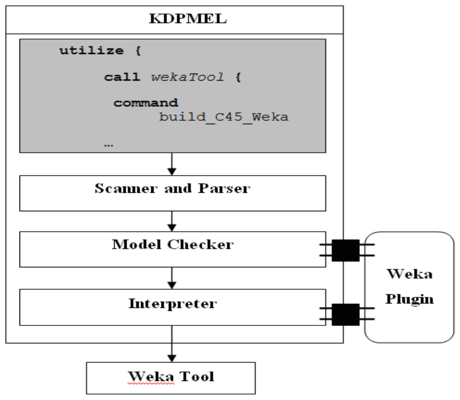

Fig. 4. KDPMEL External Commands Plug-in Mechanism

We believe that this is a novel approach that allows defining both generic and specialized KDD tasks at different levels of details and supports the goals of KDPMEL.

Task States and Transitions

The states and transitions of KDPMEL tasks are implemented using the State Pattern [14]. Tasks within a KDPMEL program go through several states during the execution of the program. The state of a task changes based on the control flow of the program, the availability of the resources needed by a task, and the explicit response from human actors. KDPMEL adopts states similar to those of Little-JIL--*posted*, *started*, *completed*, *terminated*, and *retracted*--- and adds the two new states *suspended* and *resumed* that have been suggested by Lee [5]. Fig. 5 illustrates KDPMEL task states and their transitions.

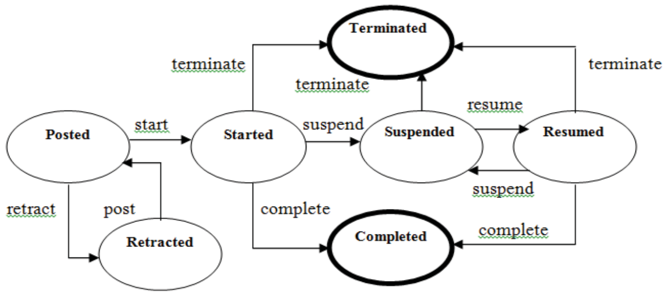

Fig. 5. Task State Transitions in KDPMEL [5]

When a KDPMEL task becomes available for execution, a task instance is created and its state is set to *posted*. A *posted* task instance is started by an explicit *start* action from the task performer (actor), which sends a start event to the task controller to change the state of the task instance to *started*. A *posted* task instance can also be temporarily retracted by a *retract* action.

A *started* task instance is completed by an explicit *complete* action. A *completed* task indicates that the task has successfully finished execution. This causes the enactment engine to continue executing the rest of the program by finding and posting the next available task. A *started* task instance can also be terminated by an explicit *terminate* action. A *terminated* task indicates an exception that caused the task not to be finished successfully. The handling of a *terminated* task varies depending on the type of control construct governing the task. A *terminated* task in a *sequence* or *loop* control construct causes the termination of the other tasks in the construct. A *terminated* task in a *parallel* control construct does not cause the termination of the other tasks in the construct. Therefore, the execution continues normally but the outcome of the terminated task is mainly affecting the status of its produced artifacts. Note that artifacts that are produced by a task are locked for exclusive use when the task is started and released when the task is finished with either *completed* or *terminated* state. A *terminated* task in a *choice* control construct causes the enactment engine to offer the construct alternatives, including the terminated task, to the actor to select a task. This approach promotes experimentation for performing KDD tasks, where an explicitly selected task can be tried first and maybe intentionally terminated later to either try another task or retry the same task if its outcomes were unsatisfactory. It also supports the interactive nature of KDD processes.

A *started* task instance can also be suspended by an explicit *suspend* action. A *suspended* task is handled in a way similar to a terminated one with some differences. In a *sequence* or *loop* control construct, the effect is the suspension of the other tasks in the construct. To continue execution, the suspended task must be *resumed*. In a *parallel* control construct, the execution continues normally but the effect of the suspended task is the continuation of locking its produced artifacts, thus preventing other tasks from starting execution because their needed artifacts are not released. This is different than a *terminated* task, where the lock is released so another task that needs to update the artifacts does not have to wait and can start execution. A *suspended* task in a *choice* control construct has the same effect of a *terminated* task and is handled similarly with the exception that it cannot be retried, because its needed artifacts are locked by the first try.

In addition to the explicit response from the actor to change the state of a task, other factors are considered before making the transition and changing the task state, such as the availability of the artifacts consumed by a task and the fulfillment of its *pre-conditions* before changing its state to *started*. Similarly, the fulfillment of the task's *post-conditions* is required before changing its state to *completed*. Another factor is the control flow of the tasks and how each control construct affects the state of its tasks. Starting a *parallel* control construct causes the states of all its tasks to be changed to *posted*. Starting a *choice* control construct causes the state of the selected task to be changed to *posted*. Starting a *sequence* control construct causes the state of the first task in the sequence to be changed to *posted*. Subsequently, the state of the next task in the sequence is changed to *posted* only if the state of current task is changed to *completed*, either from *started* or *resumed*. A *loop* control construct is handled similarly with the difference that the state of the first task in the loop after the first iteration is changed to *posted* if the requirements of the first task following the loop are not met.

3 The KDD Process-Centered Support Environment (PCSE-KDD)

PCSE-KDD is an Integrated Development Environment that is built around KDPMEL, with an IDE-style approach to facilitate the development, execution, and management of KDPMEL programs. The environment aims to provide effective management for the KDD process by supporting its entire lifecycle, and offering a variety of services, similar to those offered by PCSEEs, but directed toward KDD and data analysis processes. Environment support includes assistance for developers, maintenance of process resources, automation of routine tasks, invocation and control of development tools, and enforcement of mandatory rules and practices. Fig. 6 illustrates the high level architecture of the environment. For additional information about PCSE-KDD, see [1].

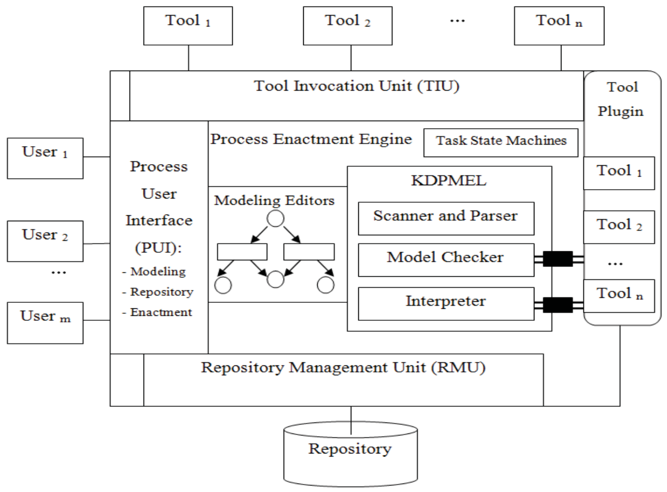

Fig. 6. The high level architecture of the PCSE-KDD

4 Evaluation and Lessons Learned

Our experience using KDPMEL to specify various aspects and tasks of KDD process supports our first hypothesis that a language-based and process-oriented approach is a flexible and effective approach to precisely and explicitly specify KDD processes as process programs that can be manipulated by programming techniques to reason about the process and support its correct execution.

KDPMEL simplicity goal achieved by having simple syntax that includes three major constructs for specifying the process resources (*artifacts*, *roles/actors*, and *tools*) and three major constructs for specifying the process tasks (*activity*, *action*, and *command*). The Language ease of use is supported by graph-based and form-based approaches to specify the various components of the process program in addition to the main source-based approach. The flexibility goal is mainly achieved by providing various levels for representing tasks at different levels of detail. Process tasks can be grouped using both strict and loose control to satisfy the highly variable control requirements of KDD processes. KDPMEL dependency control constructs allow controlling task dependencies at runtime, as opposed to static structuring in the program. The generality goal is mainly achieved by not making KDPMEL and PCSE-KDD bound to any particular KDD process models, techniques, or tools. This is accomplished by utilizing process technology to separate the KDD process definitions, which are represented as process programs, from their supporting environment, and by providing mechanisms for integrating these definitions with the environment using various programming techniques such as interpretation and validation. In addition, KDPMEL allows for specializing KDD tasks using the *command* construct and a plug-in mechanism that expand the environment with arbitrary external tools. The expressiveness goal is achieved by providing various constructs in KDPMEL to accurately reflect the KDD process by representing the tasks and their sequencing and dependencies, the artifacts consumed and/or produced by the tasks, the tools utilized by the tasks, and the actors performing the tasks.

During the implementation of a non-trivial KDD process [1], we were able to describe the process tasks and resources at different levels of detail because KDPMEL allows for specifying its constructs at arbitrary levels of detail. We were also able to execute the same process with minor variations to mine over different subsets of the data and to experiment with different tasks.

With regard to representing and integrating specialized KDD tasks, we have struggled between balancing the expressiveness and generality goals of KDPMEL. On one hand, the expressiveness goal suggests that the language should be able to accurately reflect KDD processes by representing both generic and concrete tasks, which requires representing particular KDD techniques and tools in KDPMEL. On the other hand, the generality goal suggests that the language should be applicable to KDD processes regardless of their particular models, techniques, and tools. The approach that we have adopted is to use a generic *command* construct along with a plug-in mechanism to translate these commands into representations that are accepted by concrete tools. We believe that this approach is quite general and supports all the language goals.

With regard to representing interactive KDD tasks that require the involvement of the task performer during the execution of the task, it was sufficient to represent these tasks using the KDPMEL generic *action* construct together with its associated *guidance* construct, which specifies informative guidance to help in carrying out the execution of the task.

Our experience using KDPMEL and PCSE-KDD to represent and execute various aspects of KDD process supports our second hypothesis that effective support and customized guidance, which depend on the concrete process itself rather than its generic process model, can be achieved by manipulating the explicit representation of the process in order to manage its various components and support its performance.

5 Conclusions

In this paper, we proposed the Knowledge Discovery Process Modeling and Enacting Language (KDPMEL) and outlined its Process-Centered Support Environment (PCSE-KDD) that can be used to develop KDD processes in a way that is similar to developing software processes, which is based on encoding KDD processes as process programs written in KDPMEL and exploited by PCSE-KDD to provide execution support and management for KDD processes.

KDPMEL provides a hybrid modeling approach for specifying KDD processes, mixing different types of editors and views in source-based, graph-based, and form-based styles to allow both technical and non-technical users to participate in the development of KDD processes. KDPMEL provides various language constructs to control task sequencing and dependencies as well artifacts consumed and/or produced, tools utilized, and the actors performing the tasks. KDPMEL allows for capturing the process tasks at different levels of abstraction to represent the process phases along with its generic and specialized KDD tasks. KDPMEL provides special control constructs that can be associated with a task to indicate its dependency requirements. This allows for explicitly representing and effectively managing dependencies among KDD techniques. KDPMEL provides the ability to explicitly model and enforce requirements and acceptance criteria of KDD tasks, which is important for making the KDD process more understandable, evaluating its correctness, assuring its consistent execution, and validating its results.

In KDPMEL, the process concept is supported and enforced according to a specialized KDD process that includes specific tasks organized according to their sequencing, dependencies, and alternatives.

6 References

[1] Mansour, H. A., Duchamp, D., and Krapp, C.-A. *A Language-Based and Process-Oriented Approach for Supporting the Knowledge Discovery Processes*. In Proceedings of the 11th International Conference on Data Mining (DMIN'15) (pp. 107-115), July 2015.

[2] Colin Shearer. *The CRISP-DM Model: The New Blueprint for Data Mining*. Journal of Data Warehousing, Volume 5, Number 4, 2000.

[3] David Jensen et al. *Coordinating Agent Activities in Knowledge Discovery Processes*. Department of Computer Science, University of Massachusetts Amherst, 1999.

[4] Noll, J. and Scacchi, W. *Specifying process-oriented hypertext for organizational computing*. Journal of Network and Computer Applications (2001) 24, 39-61, 2001.

[5] Lee, H. *Evaluation of Little-JIL 1.0 with ISPW-6 Software Process Example*. Department of Computer Science, University of Massachusetts, Amherst, MA 01003, March 1999.

[6] L. Kurgan and P. Musilek. *A Survey of Knowledge Discovery and Data Mining Process Models*. Knowledge Engineering Review, 21(1), pp. 1-24, 2006.

[7] Marban, O., Mariscal, G., Menasalvas, E., and Segovia, J. *An Engineering Approach to Data Mining Projects*. Intelligent Data Engineering and Automated Learning – IDEAL 2007, LNCS 4881, pp. 578-588, 2007.

[8] Marban, O., Segovia, J., Menasalvas, E., and Fernndex-Baizn, C. *Toward data mining engineering: A software engineering approach*. Information Systems 34 (1), 2009.

[9] OMG, Inc. UML Infrastructure Specification. URL: http://www.omg.org/technology/documents/formal/uml.htm. Version 2.0, March, 2006.

[10] OMG, Inc. Common Warehouse Metamodel (CWM) Specification. URL: http://www.omg.org/technology/documents/formal/cwm.htm. Version 1.1, March 2003.

[11] Greg Nordstrom et al. *Metamodeling - Rapid design and evolution of domain-specific modeling environments*. IEEE Engineering of Computer Based Systems (ECBS), Nashville, TN, April 1999.

[12] DePaul University, Chicago, IL. Classification via Decision Trees in WEKA. URL: http://maya.cs.depaul.edu/classes/ect584/WEKA/classify.html

[13] University of Waikato, New Zealand. Weka 3: Data Mining Software in Java. URL: http://www.cs.waikato.ac.nz/ml/weka/. Version 3.6, 2010.

[14] The State Machine Compiler (SMC) Framework. URL: http://smc.sourceforge.net/

[15] The Visitor Design Pattern. URL: http://sourcemaking.com/design_patterns/visitor

[16] OMG, Inc. Software Process Engineering Metamodel Specification. URL: http://www.omg.org/technology/documents/formal/spem.htm. Version 1.1, January, 2005.