Coordinating Agent Activities in Knowledge Discovery Processes

**David Jensen, Yulin Dong, Barbara Staudt Lerner, Eric K. McCall,
Leon J. Osterweil, Stanley M. Sutton, Jr., and Alexander Wise**
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{jensen|yldong|lerner|mccall|ljo|sutton|wise}@cs.umass.edu

Abstract

Knowledge discovery in databases (KDD) is an increasingly widespread activity. KDD processes may entail the use of a large number of data manipulation and analysis techniques, and new techniques are being developed on an ongoing basis. A challenge for the effective use of KDD is coordinating the use of these techniques, which may be highly specialized, conditional and contingent. Additionally, the understanding and validity of KDD results can depend critically on the processes by which they were derived. We propose to use process programming to address the coordination of agents in the use of KDD techniques. We illustrate this approach using the process language Little-JIL to program a representative bivariate regression process. With Little-JIL programs we can clearly capture the coordination of KDD activities, including control flow, pre- and post-requisites, exception handling, and resource usage.

Keywords: Knowledge discovery process, Knowledge representation, Agent coordination, Agenda management, Process programming
Category: Speculative paper

1. Introduction

KDD—knowledge discovery in databases—has become a widespread activity undertaken by an increasing number and variety of industrial, governmental, and research organizations. KDD is used to address diverse and often unprecedented questions on issues ranging from marketing, to fraud detection, to Web analysis, to command and control. To support these diverse needs, researchers have devised scores of techniques for data preparation, transformation, mining, and postprocessing. Moreover, dozens of new techniques are added each year. While the growing collection of techniques and tools helps address the growing set of needs, the size and rapid growth of the collection is becoming something of a problem itself. Many of the techniques will yield incorrect results unless they are used correctly with other techniques. In addition, KDD is often done by teams whose activities must be correctly coordinated.

Thus, one of the chief challenges facing an organization that wishes to conduct KDD is in assuring that data analysis and processing techniques are used appropriately and correctly and that the activities of teams assembled to do KDD are properly controlled and coordinated. The applicability of techniques can depend on a number of factors, including the question to be addressed, the characteristics of the data being studied, and the history of processing of those data. This problem can be compounded if the organization lacks experience with the (possibly new) techniques, or if individual analysts on a team differ with respect to their general level of expertise, specialized knowledge about the data (e.g., biases and assumptions), or familiarity with particular analysis techniques (pitfalls and tricks). The problem can be further exacerbated if multiple analysts must be orchestrated in a KDD effort, or if the resources required to support the KDD effort are scarce or subject to competitive access.

We view these problems as issues of coordination, with the general goal being to assure that the right team member applies the right technique to the right data at the right time. Similar problems of coordination come up in software development, for example, in the application of software tools to software artifacts, the assignment of developers to development tasks, and the organization of tasks in the execution of software methods. We have applied process programming to solve coordination problems in software development, and we believe that process programming is also suited to representing and supporting coordination in KDD processes. The applicability of approaches based on software process programming is further suggested by other similarities between KDD processes and software processes. For example, both sorts of problems entail the involvement of both human and automated agents, the combination of algorithmic and non-algorithmic techniques, the reliance on external resources, and the need to react to contingencies and handle exceptions. Additionally, issues of process are important in understanding and assuring the validity of KDD results.

In this paper we argue that a process orientation is important for KDD and that process programming is an appropriate technique for effecting good coordination in the use of KDD techniques. We support this argument with examples programmed in Little-JIL, a process language that emphasizes coordination of activities, agents, and the use of resources and artifacts. We believe that Little-JIL provides a basis for orchestrating coordination that assures correctness and consistency in the specification and execution of KDD processes, and assures that agents will have the ability to communicate, analyze, and generally reason about the coordination of KDD techniques.

2. KDD Processes

A process can be thought of as a multi-step plan for completing a given task. A process specification defines a class of process instances. Each instance conforms to the specification, but carries out its work in ways that are molded by the mix of agents and data that are available when the process is executed. Instances differ from each other in ways that include the selection of agents that execute particular steps, the order in which steps are executed, and the choice of which substeps are used to complete a given step.

For example, a single KDD process specification for bivariate regression might allow choice among multiple methods for handling outliers (e.g., manual removal, automatic removal, non-removal), for constructing a regression model (simple least-squares regression, locally-weighted regression, and three group resistant line), and for estimating statistical significance (parametric estimates, randomization tests). Naively assuming no interstep constraints and only these three steps, this very simple process can be instantiated in 18 different ways — a potentially confusing number for an unaided user.

Some of these possible configurations of process steps are clearly more desirable and effective than others in different situations. Thus researchers and practitioners have begun to provide this sort of guidance. Presently this takes the form of technical papers that specify desirable processes in informal ways. We believe that there is considerable value in augmenting these informal descriptions with the more precise, complete, and formal specifications that are achievable through process programming. Capturing and representing processes precisely, completely, and clearly is notoriously difficult, but our preliminary work indicates that carefully designed process specification languages can greatly facilitate this task.

2.1 Processes are Particularly Important to KDD

Explicit representation of processes is particularly important in KDD. First, effective KDD requires managing dependencies between steps. Some steps may require, disallow, or enable other steps. For example, using most neural network training algorithms requires a preceding step to recode missing values. Non-parametric regression techniques disallow any subsequent step to construct parametric confidence intervals. Constructing a decision tree enables a future step of pruning that tree. Explicit representations of these dependencies can assure that they are appropriately handled.

Second, the details of processes are essential to determining the statistical validity of inductive inferences. One example of this is the well-known error of testing on training data (Weiss and Kulikowski 1990). KDD processes that do not enforce separation between training and testing data (e.g., through simple disjoint sets or cross-validation) will produce biased estimates of model accuracy. The underlying cause of this phenomenon — referred to as "multiple comparisons" in statistics — has far more general effects. It has been causally linked to several pathologies of data mining algorithms, including attribute selection errors, overfitting, and oversearching (Jensen & Cohen 1997) and pathological growth in the size of decision trees (Jensen, Oates, and Cohen 1997). It has also been causally linked to errors in evaluating several types of modeling algorithms (Feelders and Werkooijen 1995; Gascuel and Caraux 1992; Giles and Lawrence 1997). KDD systems that employ multiple analysts distributed in time and space are particularly susceptible to pathologies stemming from multiple comparisons (Jensen 1997). Explicit representation of KDD processes supports analyses that can determine when these pathologies can and cannot occur. In addition, the ability to reinvoke an identical process is a necessary prerequisite to solutions such as randomization tests, cross-validation, and bootstrap estimates (Noreen 1989). Explicit representation of processes provides a vehicle for assuring that reinvocations are indeed identical.

Third, process details are vital to establishing the validity of KDD results in more general ways. The literature of KDD, statistics, and machine learning is filled with discoveries of implicit assumptions underlying particular techniques. In most cases, the only way to verify whether these assumptions are met is to examine the process used to apply a particular technique. Only by knowing the process used to derive a result can potential errors be traced back to their source. Explicit KDD process descriptions capture these details.

Fourth, explicit representation of KDD processes can help balance multiple performance goals. Several approaches to a given analysis task may produce results of differing statistical validity, comprehensibility, and ultimate utility. In addition, those techniques may require different amounts of computation effort and human attention. By explicitly representing these characteristics as part of the specification of individual steps, the process specification can be created that meets particular objectives (e.g., "give me a fast approximate result" or "give me a highly accurate result, but take all night if you need it").

2.2 Combining Human Analysts and Automated Agents

Research on KDD processes represents a return to one of the central issues of early work in KDD: how best to combine the goals and expertise of human users with powerful automated data analysis tools. While this topic was identified as a central one by early work in the field (e.g., Gaines 1991), it can be overlooked in our rush to develop more sophisticated automated techniques. Recent work has returned to this theme, including general descriptions of KDD processes (e.g., Fayyad, Piatetsky-Shapiro, and Smyth, AI Magazine, 1996), analysis and integration of steps (Engels, Lindner, and Studer 1997; Zhong et al. 1997), formulation of exploratory data analysis as an AI planning activity (St. Amant and Cohen 1997), and a nascent industry effort to formulate standard KDD processes (CRISP-DM). More broadly, we believe that the effective integration of the work of human and automated agents is a problem that is at the core of a growing number of critical problems. We believe that we can advance work on this problem by studying it in the more specific context of mixed-agent coordination in KDD process specification.

One important note: our work explores how to coordinate the activities of multiple KDD agents, be they automated or human. Our work does not concern programming individual automated agents for such tasks as training a neural network or calculating a chi-square statistic. These tasks are best done using conventional programming languages and software engineering techniques. Our work also does not attempt to tell human analysts how to do their job. Human analysts have knowledge and expertise that is essential to the KDD process. Instead, we are exploring flexible languages that can be used to coordinate the actions of experienced human analysts with those of automated agents and to build processes that enable less experienced analysts to achieve high-quality results. The next section provides an extended example of one such language.

3. An Example: Bivariate Regression

In this section we present an example of a KDD process for bivariate regression. Regression appears to be a relatively simple process, but it is an appropriate example nevertheless. First, it is a common data analysis activity, regression tools are included in several KDD workbenches, and it is a basic task in deployed KDD applications. Second, the process is not actually as simple as it may appear. It involves a combination of human and automated agents, it may draw on a variety of analytical techniques, the use of these techniques may be conditional and contingent, interdependencies exist between certain techniques, and the whole process may entail sequential, parallel, alternative, and recursive activities. Thus, although bivariate regression is a relatively "small" process, it still suffers many of the coordination problems that process programming is intended to address.

The basic bivariate regression problem can be described simply (see Figure 1a). We have a continuously-valued variable X (e.g., advertising spending), and we wish to determine whether it can help us predict another continuously-valued variable Y (e.g., net sales). To assess this relationship between X and Y, we have a data sample of N (x, y) tuples.

In this section, we present a process that coordinates agents and techniques in the performance of bivariate regression. We begin with basic linear regression, and then expand the example to incorporate further functionality in the form of non-linear regression and accommodation of inhomogeneous data sets (i.e., data reflecting two or more independent phenomena). The process is defined using the Little-JIL process language (Wise 1998), which is described with reference to the examples.

This process should not be taken as a complete or comprehensive specification. It contains both intentional and unintentional simplifications. That said, we believe that it illustrates many of the necessary features of a more complete specification, and that the Little-JIL language could be used to represent many of the necessary details in a more complete specification.

3.1 Linear Regression

The most common approach to the task of bivariate regression is linear regression. Linear regression constructs a model of the form $y = ß_1 x + ß_0$, and allows easy assessment of the statistical significance of the slope $ß_1$. We can conclude that X and Y are dependent if we can reject the null hypothesis that $ß_1$ is zero with high confidence.

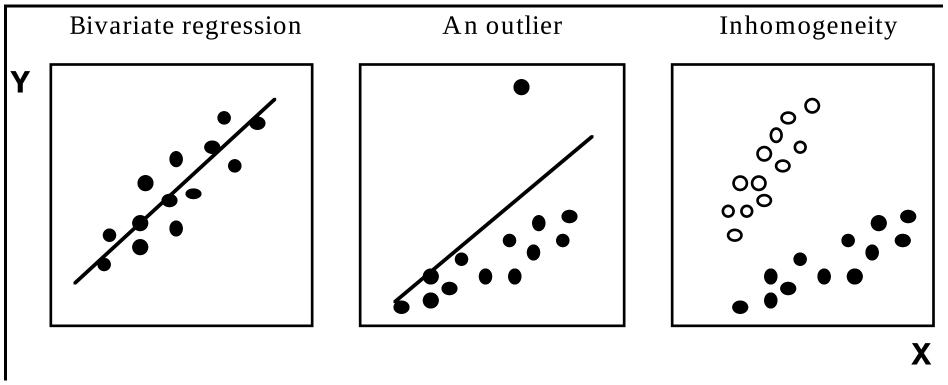

Figure 1: Simple bivariate regression and two common problems

Least squares regression (LSR) is the most commonly used form of linear regression. The advantages of LSR include relatively high statistical power and computational efficiency. However, LSR's desirable characteristics rest on several assumptions, including *homoskedasticity* (the variance of Y is independent of X) and the absence of *outliers*— (x, y) tuples that lie far from all other points. Outliers often represent errors or highly unusual conditions that produce extreme values.

Consider the assumption about outliers in more detail. Outliers strongly affect LSR models—a single outlier can sharply shift an LSR model, causing it to accurately predict neither the outlier, nor the other data points (Figure 1b). An alternative modeling technique —three group regression (TGR) (Emerson and Hoaglin 1983)—is robust to the presence of outliers. TGR divides the range of X into three groups with equal numbers of points, finds the median X and Y value of each group, and constructs a line from those three points. Because the median is a measure of central tendency that is resistant to outliers, TGR is much less strongly affected by outliers than LSR.

TGR addresses the problem of outliers, but the parametric significance test of $ß_1$ used for LSR does not apply to TGR. Instead, a computationally-intensive technique—randomization test (Cohen 1995, Edgington 1995)—should be used to test significance for $ß_1$, the slope of the line built with TGR. Incidentally, a randomization test can also be used for LSR (although, due to its computational cost, we chose to exclude this from our example process).

How the varied activities of linear regression should be coordinated, in light of the relevant dependencies, conditions, alternatives, and contingencies, is precisely what a cogent process definition should make clear. Such process definitions require a process language that enables coordination semantics to be expressed clearly and concisely, that allows rigor and flexibility to be combined as appropriate, and that supports effective process enforcement while admitting dynamic adaptation.

3.2 Representing a Linear Regression Process

In this section we illustrate the linear regression process using the Little-JIL process language. Little-JIL is a visual language derived from a subset of JIL, a process language originally developed for software development processes (Sutton and Osterweil 1997). Little-JIL focuses on coordination of agents in the performance of process activities in a wide range of processes.

Little-JIL represents the activities of a process as steps, where each step can be decomposed into substeps. Substeps within a step can be invoked either proactively or reactively. A step may also have a prerequisite to guard entry into the step, a postrequisite to guard exit from the step, and exception handlers to handle exceptions thrown during the step. The requisites and exception handlers in turn are steps that may also have substeps, etc. In addition, steps may include resource specifications. Runtime management of resource allocation provides another means of dynamically constraining, adapting, and controlling process execution. Each step also has, as a distinguished resource, an execution agent, which is responsible for initiating and carrying out the work of the step. Execution agents may be human or automated, and both types may be transparently combined in a Little-JIL process. These features and others are illustrated and discussed below with respect to the examples.

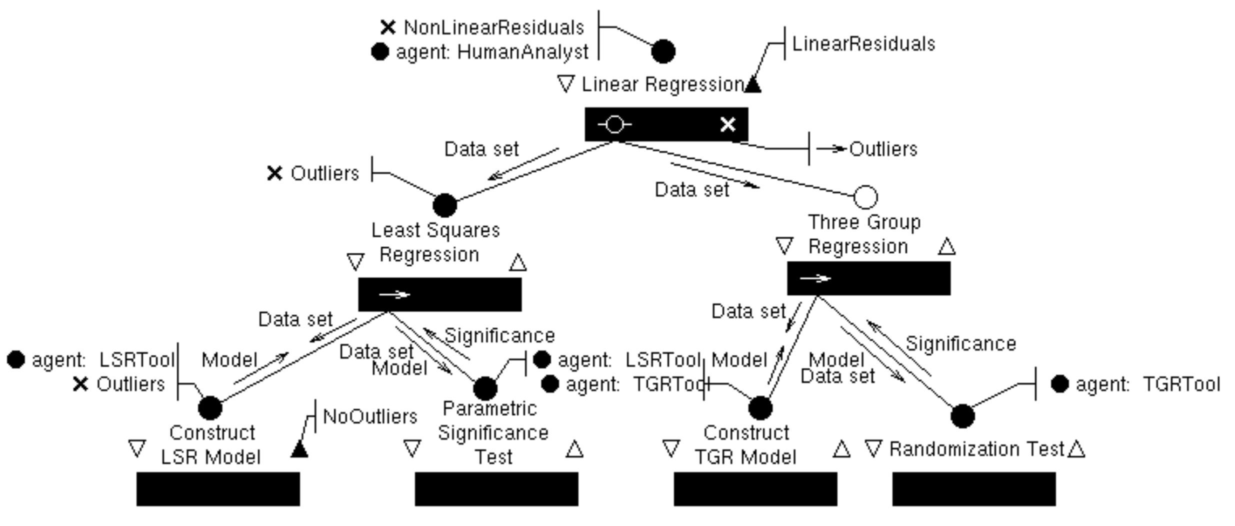

Figure 2: Little-JIL specification for linear regression

Figure 2 shows a Little-JIL specification of a linear regression process. Process steps in Little-JIL are represented visually by a step name surrounded by several graphical badges that represent aspects of step semantics. The bar below the step represents control of substeps. The leftmost element in the control bar is a sequencing badge that indicates how substeps should be executed. For example, the `Linear Regression` step in Figure 2 contains a circle-with-slash badge that represents a "choice" control construct; this indicates that `Linear Regression` is executed by executing one of the alternatives `Least Squares Regression` or `Three Group Regression`. The agent, an analyst to whom the step is assigned, makes this choice. `Least Squares Regression` and `Three Group Regression`, in turn, are executed by executing a sequence of substeps, as indicated by the arrow control badge. (Two other proactive control badges, "try" and "parallel", are discussed with respect to later figures.)

The rightmost element of a step control bar represents exception handlers. Exception handlers may be simple actions or more complex subprocesses, represented by additional substeps. The simple actions include completing the step, continuing the step, restarting the step, and rethrowing the exception. In Figure 2, the exception handler for the `Outliers` exception (thrown by step `Construct Linear Model`) has no substep; rather, this handler simply traps the exception and continues the `Linear Regression` step, as indicated by the arrow badge associated with the exception handler. (A handler with a substep is shown in Figure 4.) In the context of a choice step, continuing after an exception means that the agent is offered a choice of the remaining alternatives. A step may also include reactions, which are attached as substeps to a badge in the center of the control bar (however, reactions are omitted here for the sake of simplicity).

In the visual representation of Little-JIL steps, a circular badge above a step name represents the interface to the step. The interface includes resources needed by the step, as well as parameters sent into and out of the step, local data, and events and exceptions that may be thrown by the step. Execution agents are represented as a type of resource. Each step has an execution agent; if none is specified for a step, the execution agent is inherited from the step's parent. In Figures 2 and 3 the agents include both humans and automated tools. Data sets can also be modeled as resources. Several steps in the example throw exceptions (designated in the interface by an X). While much of the data flow between steps is shown in a simplified form, most of the data declarations have been omitted from the interfaces in the figures for the sake of brevity.

A Little-JIL step may also have a prerequisite and/or a postrequisite. A prerequisite is indicated by a downward-pointing triangle on the left of the step name and a postrequisite is indicated by an upward-pointing triangle on the right. An empty triangle indicates no requisite; a filled triangle with text indicates the name of the specified requisite. The body of the requisite is a separately specified step (not shown in our figures) possibly containing multiple substeps. A requisite is successful if it terminates normally; if it fails, it throws an exception. For example, the step `Construct LSR Model` has the postrequisite `No Outliers`. If outliers exist, then the postrequisite throws the `Outliers` exception, which causes `Construct LSR Model` to fail. The parent step `Least Squares Regression` propagates the exception, which is handled by its parent `Linear Regression`.

Clearly, there are many ways to add to the process specified in Figure 2. Additional pre- and post- requisites could be added to the LSR and TGR steps, data preprocessing steps could be added to improve the robustness of the process, and other approaches to regression could be added. The next section discusses one of the most important elaborations to the process: how to deal with non-linearity.

3.3 Coping with Non-linearity

A common diagnostic technique for any form of linear regression is to examine a plot of residuals. Ideally, the residuals—the errors in Y left unexplained by a model—should not vary with X. A non-linear relationship between X and the residuals indicates a non-linear relationship between X and Y, one that is not adequately captured by the linear model. Checking for linear residuals can be represented in Little-JIL as a postrequisite for the `Linear Regression` step. What if this postrequisite fails? One solution would be to try a non-linear modeling technique such as locally-weighted regression or lowess (Cleveland and Loader 1995). Figure 3 shows a process that includes both the original linear regression step and a new step for non-linear regression. The "try" sequencing badge on the root regression step indicates that non-linear regression is invoked only if linear regression fails. Given the current specification of linear regression, the principal reason the step might fail is the presence of non-linear residuals.

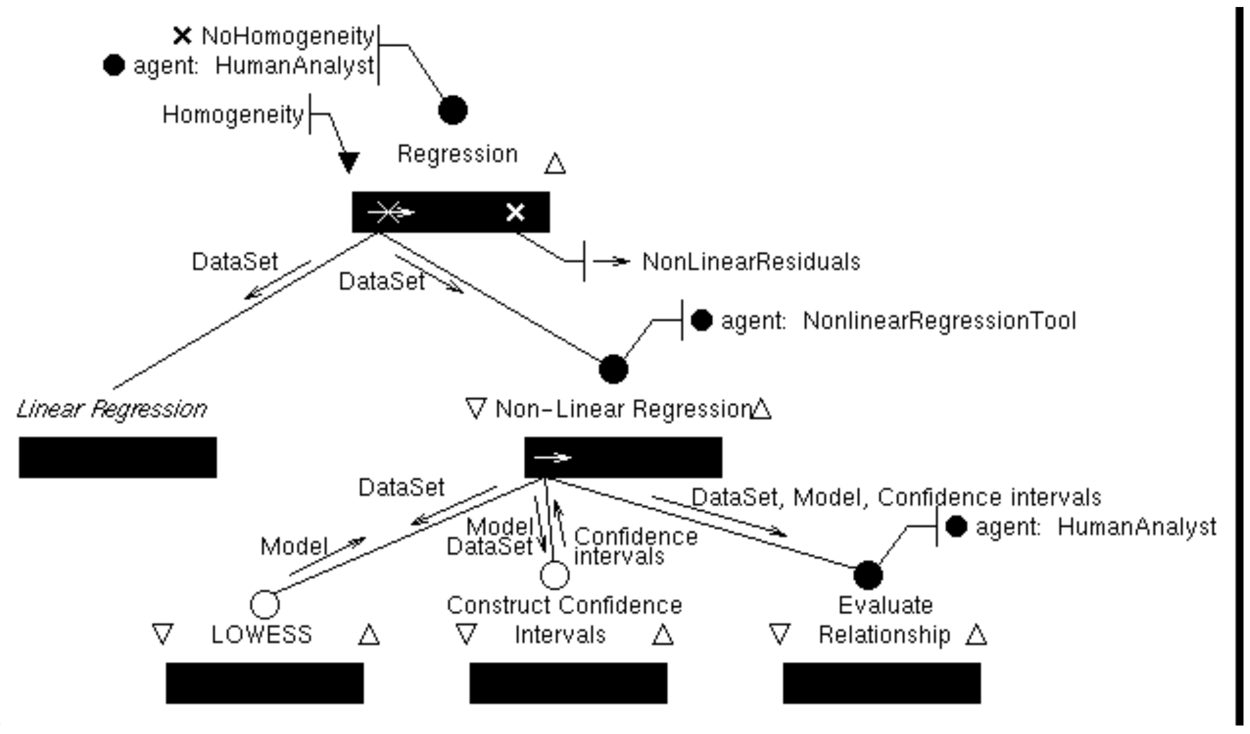

Figure 3: Regression with substeps for linear and non-linear regression

Linear regression and non-linear regression are partitioned as separate alternatives because different processes are required to determine if linear and non-linear models indicate a relationship between X and Y. Linear regression tests a relatively simple statistical hypothesis ($ß_1 = 0$); non-linear regression relies on a step `Evaluate Relationship` in which a human analyst makes a qualitative judgement. To assist in that judgment, a step to construct confidence intervals has been added to non-linear regression, although analysts should be cautious to distinguish between confidence intervals and significance tests (Cohen 1995).

Note that the overall `Regression` process coordinates the work of human and non-human agents who participate at various levels in the process. As with linear regression, many additions to the regression process are possible. These include additional approaches to non-linear regression, more quantitative substitutes for the evaluate relationship step, and prerequisites for the regression step. The next section describes one particularly important prerequisite for regression—homogeneity.

3.4 Coping with Inhomogeneity

A frequently overlooked assumption of regression is that the data sample is *homogeneous*—that it represents a single uniform phenomenon rather than two or more phenomena with fundamentally different behavior (Figure 1c). For example, inhomogeneity can occur when men and women have different physiological responses to some phenomenon, yet data from men and women are mixed together for purposes of analysis. In contrast to outliers, which often represent errors that cannot be explicitly modeled, inhomogeneity represents two or more distinct data regimes that require independent modeling.

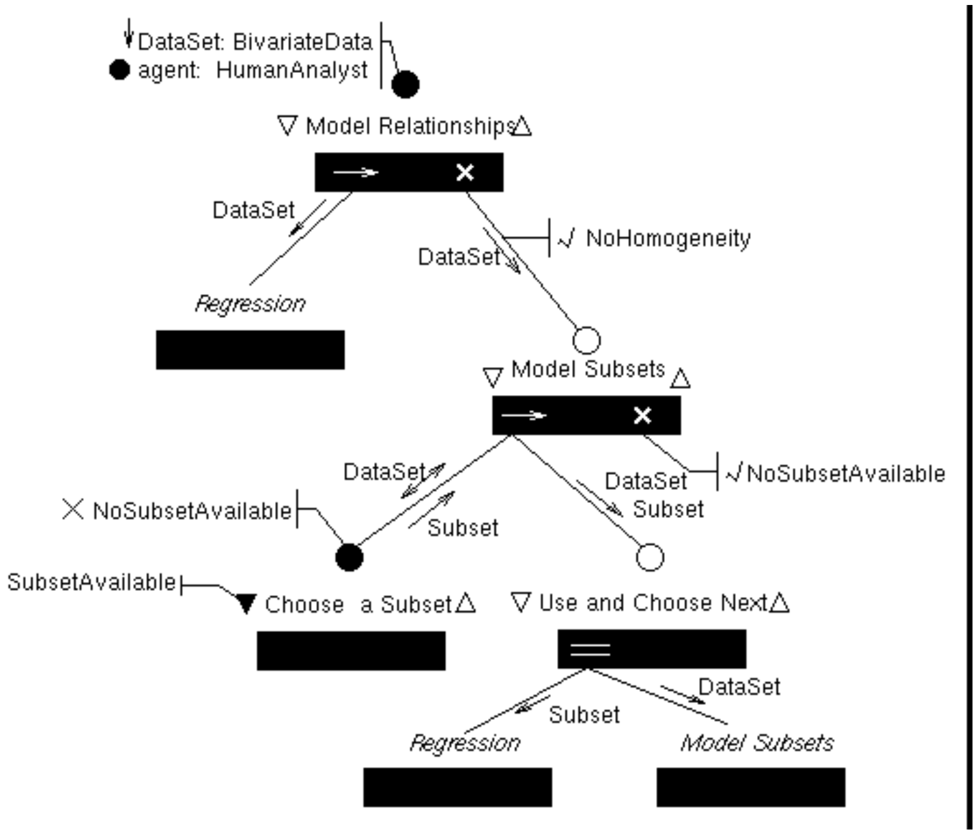

Figure 4: Handling inhomogeneous data

Figure 4 shows a `Model Relationships` process that handles inhomogeneous data. The process first attempts to apply regression testing to a given bivariate data set. However, the regression step is guarded by a prerequisite that tests the homogeneity of the data. This prerequisite assures that a single regression is not performed on heterogeneous data. If the prerequisite is violated, the exception `NonHomogeneity` is thrown, which is caught by an exception handler for `Model Relationships`. The recursive process `Model Subsets` handles the exception. At the top level `Model Subsets` is a sequence. The first substep, `Choose a Subset`, chooses a data subset from the inhomogeneous data set. The second substep is a parallel step, `Use and Choose Next`. This substep, in parallel, applies regression to the selected subset and recursively calls `Model Subsets` on the remaining part of the data set. By this recursion, `Model Subsets` iteratively models subsets of the original data set, completing normally when no more subsets are available (as indicated by the "check" badge on the exception handler for the exception `NoSubsetAvailable`).

By combining the parallel step with recursion, multiple data subsets may be modeled concurrently. Note that, in this formulation of the process, a chosen data subset is not guaranteed to be homogeneous. In that case, when the process `Regression` is called on the subset, the homogeneity prerequisite will again throw the `NonHomogeneity` exception, which will take control back to the exception handler for inhomogeneous data (i.e., `Model Subsets`). As an alternative, we could have put a test for homogeneity as a postrequisite on the `Choose a Subset` step.

3.5 Coordinating Agents at Process Execution Time

In the preceding sections we have shown how Little-JIL can be used to flexibly specify a process that manages inter-step process dependencies for multiple execution agents. In this section, we describe how the activities of these agents are coordinated when a process is instantiated and executed.

The vehicle for agent coordination during process execution is an agenda management system (AMS). An agenda management system is a software system that is based on the metaphor of using agendas, or to-do lists, to coordinate the activities of various human and automated agents. In such a system, task execution assignments are made by placing agenda items on an agenda that is monitored by one or more execution agents. Different types of agenda items may be used to represent different kinds of tasks that an agent is asked to perform.

Our agenda management system (McCall, Clarke, Osterweil 1998) is composed of a substrate that provides global access to AMS data, a set of root object types (agendas, agenda items, etc.), application-specific object types that extend the root types, and application-specific agent interfaces (e.g., GUIs for human agents).

We have designed and implemented an AMS specifically to support the execution of Little-JIL processes. This AMS has five types of agenda items: one item type corresponds to each of the four Little-JIL step kinds, and one item type corresponds to a process step at its lowest level of decomposition. Each Little-JIL agenda item has many attributes, including step name, execution agent, current status, log, step instance parameters, throwable exceptions, and interpreter. The last attribute is provided because, as we illustrate below, the Little-JIL interpretation architecture allows each step to have its own interpreter instance.

When a step of a process program is first instantiated, an agenda item of the appropriate type is created and its attribute values are set accordingly (e.g., status is set to “Posted,” input parameters are given the correct values). As the process executes, the attribute values change accordingly (e.g., the execution agent sets output parameter values, status is changed). Thus, process program execution state is stored within the AMS. This approach to storing process state is similar to that used in the ProcessWall (Heimbigner 1992).

An agent typically monitors one or more agendas to receive tasks to perform. Multiple agendas are used because an agent may frequently be involved in several disjoint processes (or acting in roles that are logically disjoint). When an item is posted to an agenda that an agent is monitoring, the agent is notified that the agenda has changed. In the case of a human agent, for example, this could result in a new item appearing in the person's agenda view window. The agent is then responsible for interpreting the item and performing the appropriate task. Agents may also monitor items individually; this gives them the ability to post an item to an agenda and to observe the item so they can react to changes in the item’s status, for example.

These mechanics are sufficient for the Little-JIL interpreter to instantiate and execute multi-agent Little-JIL process programs. By examining the state of an agenda item corresponding to a step of the process program, the interpreter can execute the process. When a new step is to be executed, the interpreter identifies the appropriate execution agent (with the help of a resource management system), creates an appropriately typed agenda item for that step, and posts it on the agent's agenda. As the agent executes the step, its updated status is reflected in the agenda item's status attribute value, which is monitored by the interpreter. As the status changes, the interpreter accordingly creates and posts substeps, returns output parameters, on successful completion of the step, propagates exceptions, on unsuccessful completion, and so on. Thus, an AMS provides language-independent facilities that allow coordination to take place, while the interpreter encodes key coordination semantics of the Little-JIL language itself. This design decouples concerns about why and when coordination should occur from concerns about how coordination should occur.

For example, consider how the process program fragment in figure 2 would be executed, supposing an interpreter had created an item to correspond to an instance of a `Linear Regression` step (a Little-JIL *choice* step). Assume the interpreter has identified a HumanAnalyst for this task (named Herman), posted it to Herman's agenda, and started the item's interpreter (which is stored in the interpreter attribute of the item). At this point the human analyst would use the GUI to change the status attribute of the `Linear Regression` item to "Starting." Its interpreter would be notified of this change and would create two new agenda items that correspond to the `Least Squares Regression` and the `Three Group Regression` sub-steps, then set the parent item's status to "Started." Because new agents are not specified for these steps, the interpreter would post them to Herman's agenda and would start interpreters for the new items. Herman's agenda GUI would render the agenda to clearly depict the subitems of the *choice* item as alternatives.

Suppose Herman then chooses to start the `Least Squares Regression` step, changing its status to "Starting." At this point both the `Linear Regression` item's interpreter and the `Least Squares Regression` item's interpreter would be notified of the change. The `Linear Regression` item's interpreter would react by setting the status of the other sub-item (`Three Group Regression`) to "Retracted." This would cause the item to disappear from Herman's agenda. Meanwhile, the `Least Squares Regression` item's interpreter would create a new item for the first substep, `Construct LSR Model`. Because the process specifies an LSRTool for that step, the new item would be posted to a particular LSRTool's agenda, then the `Least Squares Regression` item's status would be set to "Started." Whatever agent was monitoring the LSRTool's agenda would then be notified that the tool's agenda has changed. This agent would extract whatever information was needed by the tool from the agenda item, set the item's status to "Starting," and would invoke the LSRTool agent. Because `Construct LSR Model` is a leaf step, the item's interpreter immediately changes its state to "Started." When the LSRTool finishes, the tool's agent would set the status of the leaf step appropriately (“Completing” if successful or “Terminating” if not), and that step's interpreter would complete the transition of the leaf step to a final state. The interpreter for the `Least Squares Regression` item would be notified that the step has changed, and, depending on its status, would start the next sequential sub-step or would terminate the parent.

As previously mentioned, Little-JIL makes no distinction between human and tool agents. Similarly, neither does the AMS. As seen in the above example, different agents interact with the AMS, and consequently with the running Little-JIL process, via customized agent interfaces. For humans, this interface may be a GUI that is used to change an item's status, signal exceptions, change parameters, etc. For COTS tools (such as the LSRTool, perhaps), this interface may be a wrapper agent that integrates the tool with the AMS, spawning the tool to perform tasks in response to agenda items being posted to the tool's agenda and reporting the results of tool execution by setting agenda item attributes (e.g., parameters, status) as required.

Our early experiences support our belief that an agenda management system provides an appropriate metaphor for coordinating interaction in mixed-agent processes such as KDD. We intend to continue experimenting with the use of Little-JIL and the AMS to facilitate coordination in such processes.

4. Lessons Learned

Our experience using Little-JIL to specify KDD processes has been instructive. Many coordination aspects of KDD processes (including examples not described here) have been easily expressed using Little-JIL. For example, one aspect well handled by Little-JIL is the highly variable control requirements of KDD processes. Conversely, KDD processes have drawn on the full range of Little-JIL control constructs. In some cases, processes require extremely strict control, and Little-JIL allows us to indicate this (e.g., by executing substeps in a specified order). In other cases, only very loose control is needed, and the language allowed us to specify this as well (e.g., by allowing user choice or parallel execution). We believe that a successful process languages for KDD must allow flexibility to program processes both strictly and loosely.

Little-JIL's pre- and postrequisites are essential to effective coordination in KDD processes. Prerequisites make explicit the assumptions that underlie a sampling or analysis technique;

postrequisites make explicit the acceptance criteria for the successful completion of a technique. The ability to make assumptions and acceptance criteria explicit is important for making a KDD process understandable, evaluating its correctness, assuring its consistent execution, and validating its results.

Similarly, the ability to represent exceptions and exception handling is critical for process robustness, reliability, and safety. In our KDD examples, exception management is also crucial in specifying process control structures. While many descriptions of KDD techniques use nearly ideal data, most practitioners who attempt to apply these techniques quickly uncover hidden assumptions, leading to exceptions in idealized process models. The ability to indicate possible exceptions, specify how they are to be handled, and direct subsequent execution, is essential to coordinating KDD efforts in real-world applications.

Resource management provides another dimension of coordination in KDD processes. Flexibility in agent coordination is afforded because Little-JIL process can be written independently of the specific execution agents to which they will be bound at run time. Additionally, the control model of the language, in conjunction with the agenda management system, allows processes to be written transparently with respect to the issue of human versus automated agents. However, runtime allocation of agents allows dynamic orchestration of agent activities and enables the dynamic adaptation of process behaviors to agent availability. Similar degrees of flexibility and opportunities for dynamic control apply to resources in general.

Our experience using Little-JIL also uncovered some deficiencies in the current version of the language. Two related issues particularly stand out. First, adding to Little-JIL process specifications is not as simple as we would like, at least in the domain of KDD. Incorporating new techniques (e.g., variable transformation or deletion of outliers) sometimes requires substantial revisions in process specifications (due at least in part to the interdependence of KDD techniques and their effects). We would prefer that Little-JIL process specifications to be robust enough that most new KDD techniques could merely be "slotted into" the appropriate place in a Little-JIL process. Changes to the Little-JIL language or development of appropriate Little-JIL style may be essential steps toward this goal.

Second, KDD is often an event- or opportunity-driven activity, and our current Little-JIL process specifications have not expressed this adequately. Exception handlers have provided useful specifications of reactions to abnormal events; based on experience in modeling software development processes, we believe that the ability to specify reactions to normal events should also be useful for KDD processes. This aspect of Little-JIL is currently under development.

5. Future Work

Our work to date with Little-JIL has convinced us of the general utility of KDD process specification. However, at least three important areas of work remain. First, additional experience with specifying KDD processes is needed. We intend to extend our existing handful of KDD processes to a larger number, and to increase the level of sophistication of those processes.

Second, while we believe that Little-JIL specifications are easy to read and write compared to more algorithmic languages, we would like the process to be extended by non-programmers. We imagine providing a more sophisticated process editor that would assist a KDD researcher by assisting with the insertion of appropriate steps with the necessary prerequisites, postrequisites, data flow and exception handling.

Finally, the Little-JIL language itself is still under development and there are a number of issues we intend to address. We are investigating integrating an AI planner (Garvey, Decker, and Lesser 1994) and resource-based scheduler (Wagner, Garvey, and Lesser 1997) with Little-JIL. Such mechanisms would allow us to schedule agents and other resources based on cost, availability for a specific time and duration, and expected quality of their results. The results from planning would help guide agents in their decision making at choice and parallel steps by identifying which substeps are most likely to satisfy the time, cost, and quality constraints for process instances.

We are also investigating the use of static analysis techniques (Dwyer and Clarke 1994) on Little-JIL processes. Specifically, we wish to prove properties of Little-JIL processes such as ordering rules (Step A always executes before Step B) and non-local dependencies (if Step A is performed, Step B is

6. Conclusions

Knowledge discovery research is developing and exploiting a diverse and expanding set of data manipulation and analysis techniques. Not all analysts, or even all organizations, can have a thorough knowledge of how to correctly and effectively combine and deploy these techniques. Process programming provides an effective means for specifying the coordinated use of KDD techniques by agents in potentially complex KDD processes. As demonstrated in this paper, KDD process specifications written in Little-JIL express requirements on individual techniques and capture dependencies among techniques. Little-JIL is a high-level process language designed to support the specification of coordination in processes; Little-JIL offers appropriate control flow constructs, pre- and post-requisites, reactions, exception handling, agent specifications, and dynamic resource bindings. Little-JIL enables explicit representation of KDD processes, allows reasoning about those processes, and supports correct execution of the processes. In turn, this enables KDD applications to produce reliable and repeatable results, which is necessary for the effective use of data mining across a wide range of organizations.

7. Acknowledgments

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137 and F30602-97-2-0032.

References

Cohen, Paul R. (1995). Empirical Methods for Artificial Intelligence. MIT Press.

Cleveland, W. S. and C. R. Loader (1995). Smoothing by local regression: principles and methods (with discussion). Computational Statistics.

Dwyer, Matthew and Lori A. Clarke (1994). "Data Flow Analysis for Verifying Properties of Concurrent Programs", ACM SIGSOFT '94: Proc. of the Second ACM Sigsoft Symp. on Foundations of Softw. Engg., December, pages 62-75.

Edgington, Eugene S. (1995). Randomization Tests. Third Edition. Dekker.

Emerson, John D., and Hoaglin, David C. 1983. Resistant lines for x versus y. In Hoaglin, David C.; Mosteller, Frederick; and Tukey, John W., editors, 1983, Understanding Robust and Exploratory Data Analysis. Wiley. 129-165.

Engels, Robert, Guido Lindner, and Rudi Studer (1997). A guided tour through the data mining jungle. Proc. of the Third Intl. Conf. on Knowledge Discovery and Data Mining. 163-166.

Fayyad, Usama, Gregory Piatetsky-Shapiro, and Padhraic Smyth (1996). From data mining to knowledge discovery in databases. AI Magazine. Fall. 37-54.

Feelders, A. and W. Werkooijen (1995). Which method learns the most from data? Methodological issues in the analysis of comparative studies. Preliminary papers of the Fifth Intl. Workshop on Artificial Intelligence and Statistics.

Gaines, Brian R. (1991). Refining induction into knowledge. Knowledge Discovery in Data Bases. Workshop Notes from the Ninth Natl. Conf. on Artificial Intelligence. 1-10.

Garvey, Alan, Keith Decker, and Victor Lesser (1994). A Negotiation-Based Interface between a Real-Time Scheduler and a Decision-Maker. Proc. of Workshop on Models of Conflict Management in Cooperative Problem Solving. AAAI.

Gascuel, O. and G. Caraux (1992). Statistical significance in inductive learning. ECAI92: Proc. of the 10th European Conf. on Artificial Intelligence. 435-439.

Giles, C. Lee and Steve Lawrence (1997). Presenting and analyzing the results of AI experiments: Data averaging and data snooping. Proc. of the Fourteenth Natl. Conf. on Artificial Intelligence. AAAI Press. 362-367.

Heimbigner, Dennis (1992). The ProcessWall: A Process State Server Approach to Process Programming. Proc. of the Fifth SIGSOFT Symp. on Softw. Development Environments. December 1992, pp. 159-168.

Jensen, David D. and Paul R. Cohen (1997). Multiple comparisons in induction algorithms. Submitted to Machine Learning.

Jensen, David D., Tim Oates and Paul R. Cohen. 1997. Building Simple Models: A Case Study with Decision Trees. In Advances in Intelligent Data Analysis: Reasoning about Data. Proc. of the Second Intl. Symp., IDA-97, pp. 211-222.

Jensen, David D. (1997). Unique Challenges of Managing Inductive Knowledge. In Working Notes of the AAAI 1997 Spring Symp. on Artificial Intelligence in Knowledge Management. 75-81. Also in: Artificial Intelligence and Knowledge Management. Collected Papers from the 1997 Workshop. Technical Report WS-97-09. AAAI Press. 25-31.

McCall, Eric K., Lori A. Clarke, and Leon J. Osterweil (1998). An Adaptable Generation Approach to Agenda Management. Proc. of the Twentieth Intl. Conf. on Softw. Engg., pp. 282-291.

Noreen, Eric W. (1989). Computer Intensive Methods for Testing Hypotheses: An Introduction. Wiley.

St. Amant, Robert and Paul R. Cohen (1997). Intelligent Support for Exploratory Data Analysis. Submitted.

Sutton, Jr., Stanley M. and Leon J. Osterweil (1997). The Design of a Next-Generation Process Language. Proc. of the Joint 6th European Softw. Engg. Conf. and the 5th ACM SIGSOFT Symp. on the Foundations of Softw. Engg.. Springer-Verlag. 142-158.

Wagner, Thomas, Alan Garvey, and Victor Lesser (1997). Complex Goal Criteria and its Application in Design-to-Criteria Scheduling. Proc. of the Fourteenth Natl. Conf. on Artificial Intelligence.

Weiss, Sholom and Casimir A. Kulikowski (1991). Computer Systems That Learn. Morgan Kaufmann.

Wise, Alexander (1998). Little-JIL 1.0 Language Report. CMPSCI Technical Report 98-24. University of Massachusetts at Amherst, Computer Science Department.

Zhong, Ning, Chunnian Liu, Yoshitsugu Kakemoto, and Setsuo Ohsuga (1997). KDD process planning. Proc. of the Third Intl. Conf. on Knowledge Discovery and Data Mining. 291-294.