Towards Coordination Patterns for Complex
Experimentations in Data Mining

Farhad Arbab\textsuperscript{1}, Claudia Diamantini\textsuperscript{2},
Domenico Potena\textsuperscript{2}, and Emanuele Storti\textsuperscript{2}

\textsuperscript{1} Centre for Mathematics and Computer Science
Science Park 123, 1098 XG Amsterdam, The Netherlands
Farhad.Arbab@cwi.nl

\textsuperscript{2} Dipartimento di Ingegneria Informatica, Gestionale e dell’Automazione “M. Panti”,
Università Politecnica delle Marche - via Brecce Bianche, 60131 Ancona, Italy
\{diamantini, potena, storti\}@di.ing.unipmn.it

Abstract. In order to support the management of experimental activities in a networked scientific community, the exploitation of service-oriented paradigm and technologies is a hot research topic in E-science. In particular, scientific workflows can be modeled by resorting to the notion of process. In this paper we present a preliminary study about the exploitation of the formal coordination language Reo for representation and verification of complex coordination patterns typically appearing in distributed experimental processes for the extraction of potentially useful knowledge from data (i.e., Data Mining). The study demonstrates the feasibility of the approach, its advantages and potential for the Data Mining field and for E-science in general. It also highlights the peculiarities of Data Mining processes that solicit for an extension of the existing process modeling/checking approach.

1 Introduction

Data Mining (DM) refers to a core of statistical and machine learning techniques for the induction of models from data. In order to generate valid and useful models, the Knowledge Discovery in Databases (KDD) methodologies codify in a high-level process schema the basic steps to perform before and after the application of DM techniques (e.g., they are described as Business Understanding, Data Preparation, Modeling, Evaluation in the CRISP standard \cite{crisp}). Concrete implementations of this schema involve the realization of many different activities partially performed by humans and partially automated by suitable tools. These activities follow a blend of collaborative and challenging work models. In a collaborative model, actors\textsuperscript{3} with complementary capabilities face a problem jointly, sharing many kinds of resources, such as data, knowledge, analysis tools and devices. On the other hand, challenging work means that actors analyze a problem while competing with each other, comparing their final results.

\textsuperscript{3} Hereafter we use the term actor whenever it is not necessary to distinguish between humans and tools.
Although many KDD platforms exist where each focuses on a relatively small local set of tools executed by a single user, real complex projects are likely to involve many users, tools produced by different organizations (e.g., research centers or commercial software houses) and data; furthermore, resources can be geographically distributed. Such distribution typically arises from the nature of organizations: often split in several departments, each with its own level of autonomy and legal and administrative boundaries, and arranged as networking units. In such cases DBA, domain experts, KDD and Data Mining specialists must interact to accomplish the goal of knowledge extraction. In addition, nowadays different organizations that collaborate in joint research projects increasingly need instruments for deep analysis of data produced by scientific experiments, while new tools are continuously developed as a result of research activities and commercial developments.

In order to support the management of collaborative/challenging experimental cycles in a networked environment, several issues must be taken into account, including sharing and integration of data and knowledge, sharing and reuse of software, integrated access to resources, management of collaborative work and operations flow which in a virtual community is the in silico equivalent of an experimental process. In our previous work we tackled the problem of software sharing and reuse in open KDD community environments, by proposing a semantic-based, service-oriented platform supporting publishing, discovery, uniform access and execution of data analysis tools provided in the form of web services [2, 3]. In this paper we consider the management of operations flow, focusing on the formalization of coordination patterns. In particular, we present a preliminary study on the exploitation of the formal coordination language Reo for representation and verification of complex coordination patterns that typically appear in distributed experimental Data Mining processes. This study demonstrates the feasibility of the approach, its advantages and potential for the Data Mining field and for E-science in general. It also highlights the peculiarities of Data Mining processes that solicit for an extension of the existing process modeling/checking approach.

The remainder of this paper is structured as follows: Section 2 discusses some relevant related work both about KDD support systems and coordination languages. In Section 3 we envisage the main issues related to KDD collaborative/distributed environment: we briefly introduce our previous work on a KDD support systems and describe the need of a language for managing coordination issues that arise among the actors that take part in a KDD process. We describe such a language in Section 4, while in Section 5 we investigate its application to our scenario through a case study; Section 6 discusses the followed approach and outlines future directions to overcome the experienced issues.

2 Related Work

The exploitation of service-oriented paradigm for distributed Knowledge Discovery can be traced to the work by Sarawagi [4], which introduced the iden
of data mining models as services with the aim of facilitating the use of such
techniques among novice users. Since then, service-oriented architectures have
been exploited as a means to develop a new generation of open distributed KDD
support systems, enabling high performance distributed computing [5, 6] and the
shift from programming to rapid development through discovery and composi-
tion/orchestration of KDD services, [7, 8, 6, 2]. The major part of these proposals
aims to show the feasibility of the service oriented approach in distributed Data
Mining, and of related languages and engines for workflow definition and execu-
tion. Scientific workflow design and enactment is especially studied in E-science.
Here, scientific workflow systems are seen as a promising means to enable col-
aboration in virtual communities, giving support to experimental design and
deployment, parameter space sweeps, dataflow and concurrency optimization [9,
10]. In [11] it is recognized that present paradigms and workflow systems lead to
recurring scientific workflow design challenges, and a workflow design paradigm
based on data assembly lines is presented. We share with this work our recogni-
tion of their motivating challenges and some common principles in our adopted
solutions (e.g., channels, glue-code or adapters). However the exploitation of a
formal coordination language is a peculiarity of the work we present in this paper.
To the best of our knowledge this is the first attempt to exploit formal methods
for workflow modeling and verification in distributed KDD and E-science.

Coordination models and languages constitute a linguistic approach to pro-
vide a systematic and semantically coherent middle-ware layer between dis-
tributed applications and the low-level communication primitives supported by
typical programming languages and operating systems. Their purpose is to sim-
plify the design, verification, and reuse of coordination and concurrency protocols
[12]. Reo is an exogenous coordination language [13]. In exogenous coordination,
no participant actor is responsible for its own coordination. This results in a
clear separation of computation activity from coordination protocols and yields
more reusable computation processes as well as concrete pieces of independent,
reusable coordination code.

3 Distributed KDD environment

Managing a distributed KDD process is not a straightforward activity, because
it entails addressing a number of functional and non-functional aspects, such as:

- heterogeneity: each tool is written in a specific language and can be accessed
  through different standards;
- integration: to provide advanced support in KDD, tools must be integrated
  to build more complex services, but each tool uses a specific syntax and data
  format, which may not be compatible the others;
- localization: each distributed tool can be executed at a specific location, but
  no registry is necessarily available for locating it;
- complexity: management of a KDD process is a complex operation because
  it is made up of many different phases; furthermore, choosing the best algo-
  rithm (among the plethora of available ones) for a specific phase in a process
and properly setting its parameters are demanding activities, especially for non-expert users;

- coordination: each actor interacts with others through a specific protocol that defines the sequence of messages to send/receive; explicit representations of such protocols are necessary to specify and manage the desired coordination of these actors.

Our ongoing project KDDVM [2] (see also http://boole.diga.unipv.it) aims at addressing the above issues. KDDVM is an open architecture for facing the design and management of a complex KDD process in a distributed/collaborative scenario. It provides users with many different functionalities for accessing and executing tools in the form of services, plus additional support for publishing/discovering services and for semi-automatically composing KDD processes. Since each KDD tool may be written in a different programming language, we generate a wrapper around it, in order to make it accessible through a common standard. Moreover, heterogeneity of interfaces leads to an interoperability issue, which we address by describing each service through an XML-based descriptor, enhanced by semantic information about the meaning of the data involved in the I/O interfaces. Among the support services, a semantic broker allows authors to publish their KDD services in a registry, while end users can perform syntactic and semantic queries for retrieving services that comply with user requirements. Finally, the semiautomatic process composer helps in generating the sequence of abstract steps that must be performed to actually solve the user problem. With respect to the complexity issue, semantics is consistently used throughout the whole system to greatly enhance its ease-of-use, by providing at the same time a user-readable explanation of the meaning of data/interfaces/algorithms and a computer-readable model for reasoning on data and supporting advanced functionalities. With this service-oriented view of KDD tools we are able to support the abstract design of KDD processes and the discovery and execution of tools in an open distributed environment. However, interaction and coordination issues arise when an abstract process is implemented in an experimental workflow. In the following subsection, we offer an analysis of such issues and their related requirements.

3.1 Interaction issues

Real KDD scenarios are highly iterative and interactive: iteration can be experimented with at the process level, by re-executing a set of phases, and at the phase level, when a single tool is executed many times with different parameters. The most common meaning of interaction refers to a human that communicates with a tool, but in distributed KDD scenarios we are interested in other kinds of interaction as well, like tool-tool and human-human. Actors interact by executing each a part of a task in a process and can compete for sending messages to other actors. The execution order of such messages, their flow of data, their collaboration in a shared setting of some steps of the process must be properly managed. To make things even more complex, in such scenarios the number of
iterations, as well as the computational demands and execution times cannot be precisely known in advance, and strongly depend on partial results obtained after each phase of the KDD process.

When a service is published on the net, only its interface is known, whereas its implementation details are hidden from the users; as such, it is straightforward to execute simple services that generate a single response after a call. However, more interactive services that send/receive messages to/from users or tools can exhibit a more complex behavior. In such cases, the process designer must know in advance the observable dynamics of the service (i.e., the protocol that describes the sequence of sent and received messages). In a distributed KDD environment managing interaction is not trivial in an open distributed environment. The following considerations sketch out some requirements for a language to properly coordinate actors:

- requirement 1: integration of behaviors. KDD tools are written by different authors and know nothing about each other; since they are not designed to be used together, we need a language for explicitly modeling an interaction protocol, which does not require modifications of the tools;
- requirement 2: independent interaction representation for reuse. KDD processes can often implement complex interaction protocols, but in many cases typical coordination patterns can be found. To help users in designing a process, thus, we need a language for representing interaction independently from the specific services that interact with one another. Such a language greatly simplifies the KDD process design by allowing the reuse of "pure protocols" once they are defined;
- requirement 3: formal verification of interaction correctness. KDD processes, due to their iterative/interactive nature, may require a long time to execute; therefore, we would like to provide process designers with some guarantees about the correctness of the interactions among the actors.

A possible solution for addressing this problem is through testing all possible executions, in order to find whether some problems may arise, like deadlock, missing synchronizations, undesirable behaviors and so forth. Although testing by simulation may help in finding bugs in a design, it may be computationally too expensive, especially for highly iterative/interactive processes. In order to face the combinatorial explosion of statespace in testing and to provide some stronger guarantees about correctness, formal verification by model checking can be used to ensure that some initial requirements, expressed through a formal language, are satisfied. To accomplish this goal, model checking performs verification on a formal model of the system, in which no ambiguity is allowed. However, most languages for managing interaction lack such a strictness and focus either on choreography or on orchestration. Since these two topics can be considered as different aspects of coordination, we rely on the coordination language Reo, which is characterized by offering loose-coupling, clear separation between computation and coordination, formal semantics, and great expressiveness.
4 Reo: a coordination language

Reo [13] offers a channel-based exogenous coordination model wherein complex coordinators, called connectors are compositionally built out of simpler ones. The simplest connectors in Reo are a set of channels with well-defined behavior. Each channel has exactly two ends (each of which can be a sink that accepts data or a source that dispenses data) and a constraint that relates the flow of data at its ends, thereby implicitly specifying a policy for the channel and its various properties. The properties of the channel that are specified in its constraint include: synchronization (either I/O operations on both ends succeed atomically, e.g., SyncDrain channel, or not), buffering (the channel has a buffer for holding data, e.g., Fifo channel), ordering (the channel has a buffer that imposes an order on the delivery of its content), computation (data transformation or filtering), and data retention/loss. Some basic channels are represented in Fig.1; for a comprehensive discussion of various channel types see [13]. In order to build complex connectors out of simpler ones, Reo introduces the concept of a node. Every channel end is a singleton node and connectors are built by joining them. The formal semantics of Reo exists in various semantic models, each serving a different purpose. One such model is the coinductive calculus of flow, which represents the behavior of connector ends as timed data stream (i.e., pairs consisting of a data stream and a time stream), while the behavior of channels is represented as relations on timed data streams, expressing which combination of streams are mutually consistent. The behavior of a connector is defined as a composition of the semantics of its constituent channels and nodes. Such a model is expressive enough to allow different kinds of reasoning, including verification, in order to state whether a connector meets the formal specifications of its behavior, given the behavior of each service in a proper formalism.

The emphasis in Reo is on connectors, their behavior and their composition, but not on the entities (e.g., services) that connect, communicate, and cooperate through them. The behavior of every connector in Reo imposes a specific coordination pattern on the actors that perform normal I/O operations through that connector, without the knowledge of those actors. This means that from a Reo point of view services are simply represented by components, i.e. channels and node. This makes Reo a powerful "glue language" for compositional construction of connectors to combine component instances and services into a software system and exogenously orchestrate their mutual interactions. Exogenous coordination models allow third parties to orchestrate the interactions

\[
\begin{array}{cccc}
  \text{a} & \rightarrow & \text{b} & \quad \\
  \text{Sync}(a,b) & \quad \\
  \text{a} & \cdots & \rightarrow & \text{b} & \quad \\
  \text{LossySync}(a,b) & \\
  \text{a} & \leftrightarrow & \text{b} & \\
  \text{AsyncDrain}(a,b) & \\
  \text{a} & \rightarrow & \text{b} & \quad \\
  \text{SyncDrain}(a,b) & \\
  \text{a} & \rightarrow & \text{b} & \quad \\
  \text{Fifo}(a,b) & \\
\end{array}
\]

Fig.1. Main Reo channels
among the actors, by dynamically composing building blocks together. Thanks to this loosely-coupled paradigm, components are highly reusable because the same set of actors can be arranged together with different connectors, producing systems with very different emergent behaviors.

Reo has a formal graphical syntax analogous to electronic circuit diagrams, which makes the design of connectors easy. The Eclipse Coordination Toolkit (ECT) is available to aid process designers in representing interactions in complex processes (see http://reo.projects.cwi.nl). It is written in Java as a set of plugins within the Eclipse platform, and it currently consists of many parts; among them: (i) graphical editors, supporting circuit design; (ii) a simulation plug-in, for simulating the coordination of the circuit; (iii) a Java code generation plug-in, that represents a service coordination model as a set of java classes; (iv) a validation plug-in, that performs model checking of coordination models.

5 Reo for coordination and verification of KDD processes

Reo can be used in parallel or distributed environments as a language for coordination of concurrent and interactive processes. In our scenario, KDD services and users are independent distributed entities that utilize Reo channels and connectors to communicate. Through Reo it is possible to explicitly express all the interactions among the actors in a KDD process: tools and GUI can be thought of as components whose I/O ports are linked to Reo nodes; as data become available in some output ports, they flow to input ports according to the semantics of the connector. Implementation details remain internal to individual actors, while the behavior of the whole system is coordinated according to the semantics of the Reo circuit.

In this Section, we use a case study through which we describe the necessary steps for designing and verifying a KDD process by means of Reo tools. The scenario of our case study refers to users who want to perform a classification analysis on a dataset. For simplicity, we focus only on iteration/interaction within a single step of the KDD process, namely Modeling. Suppose that two users want to collaboratively setup a tool for generating a good model for classification. Since no general theory is able to suggest to them how to set the parameters of the various tools according to the statistical characteristics of the dataset, they are likely to try a range of values for each tool parameter. Moreover, since they may not be experts in a specific algorithm, we would like to automate the process as much as possible. In summary, the process will have to manage iterative executions of two instances of the same tool with different values, coordinating iteration and interaction with the users, while it manages concurrent access to shared resources.

5.1 Process Design

The actors that take part in a process are two users, who interact through a GUI for sending/receiving data and messages to the circuit, and four tools:
DataMiningAlgorithm tool for learning the model (in: learning set, two integer parameters; out: model);
- Test tool for testing the model (in: model, test set; out: accuracy);
- Splitter for dividing the user dataset into two parts (in: dataset; out: learning set, test set);
- SequenceGenerator tool for generating a sequence that goes from min to stop value, with a given step (in: min, stop, step; out: a value).

Following a top-down strategy in coordination engineering, we sketch out the top-level connectors that interact\(^4\), and then we realize each of them separately, using the tools as components. At a topo level, we define the KDD Experiment connector, which coordinates the interaction of the user with the whole process. Figure 2 shows the KDD Experiment connector, as composed of 4 sub-connectors. The detailed behavior of each sub-connector is as follows: Learning uses Splitter, DataMiningAlgorithm and Test tools for generating a result (model and its accuracy) by elaborating upon a given dataset and a pair of the two parameters, Par_Setting takes some parameters and generates all possible combinations of such parameters, for feeding into the algorithm. Such a behavior is realized by interconnecting two instances of SequenceGenerator in such a way that the output of the first triggers the input of the second. Model_storage stores the best model at any iteration and allows the user to read it at any time, while Control_circuit manages iteration and synchronizes the reset signal for all components, in order to bring the system to the initial state.

\(^4\) The graphical modeling tool in the ECT framework was used in the design phase.
Due to space limitation, we cannot describe in full detail all connectors; instead, we focus here only on the Model Storage connector as shown in Fig.3. Note that its behavior emerges from the composition of the semantics of the channels and nodes used in its construction, with no external tool involved. In particular, the two SyncDrain channels with the additional expression are used for discriminating between incoming models with better or worse accuracy compared with the stored one.

The whole system consists of two Experiment connectors, each for one of the two users. They are interconnected so that each user can manage his/her own experiment, but each result is provided to an external Model Storage, which stores the best model amongst them. In this way, users can read the best model of both their experiment and the collaborative experiment. Once users know which is the best parameter range, they can perform subsequent iterative executions of the experiment with a smaller range, trying to locate a local maximum (i.e., a model with a high level of accuracy). Many of the defined connectors are reusable in different configurations. For instance, Model Storage that is used inside Experiment and also as a shared resource. Furthermore, once the behavior of each component is known, it is straightforward to reuse it, because the designer is required only to put some channels for defining the needed interconnections.

5.2 Model checking

By means of the Simulator tool in the ECT framework it is possible to generate several animations to test the circuit. Although this can give hints about the correctness of the design, model checking is needed in order to have strict guarantees about the behavior of the connectors and systems. Since each Reo element is provided with a formal semantics, the model for the whole circuit is automat-
Fig. 4. LTS for the Experiment connector

ically synthesized, and we can check it, e.g., using the mCRL2 plug-in available within the ECT. mCRL2 (http://www.mcrl2.org/) is a specification language that can be used to specify and analyze the behavior of distributed systems and protocols. It is based on the Algebra of Communicating Processes (ACP) but is extended to include data and time. Given a process and a formula that expresses some desired behavior of the process, mCRL2 can decide whether the formula holds on (the behavior of) the process or not. As such, it is possible to check, once represented in a formal language, all desired behavioral properties, such as: safety properties, which guarantee that the circuit does not exhibit some undesirable behaviors (e.g., deadlock), and liveness properties, which state that some actions will always be followed by some reactions (e.g., compliance with a specific interaction protocol). To investigate the properties of the behavior of the system, such a tool generates a Labeled Transition System (LTS) of the model, i.e., a labeled directed graph representing in which states the system can be and, for each state, the interactions that are possible out of that state. For example, in Fig.4 the LTS of Experiment is shown, in which we have bounded the possible values of a model’s accuracy to the symbolic set \{0, 1, 2\} in order to avoid a state explosion. Note that the requested properties stated in Subsection 5.1 are satisfied: there is no deadlock/livelock, the Model\_storage behavior lets the user consume only the best model at any time, in a monotonic way as better models are generated. In addition, the stop signal resets the whole system to its original state.

6 Discussion

From the modeling exercise partly described in this work we can draw the following conclusions. By using Reo, the main benefits that we envisage for the management of KDD processes are the following:

- arbitrary complex interaction can be represented by means of channels and nodes; given the high expressiveness of the language, many workflow patterns
that cannot be represented by the main workflow languages can be easily implemented;

- the whole circuit, as well as its subparts, can be formally verified in order to check whether they comply with the initial requirements;

- coordination can be designed through software engineering methodologies: given the loose coupling between interaction and computation, it is possible to define connectors hierarchically, in order to enhance their modularity and reusability. As a matter of fact, it is possible to use top-down / bottom-up approaches to define connectors, each of which implements a specific interaction protocol, that in turn can contain or be contained in other connectors.

Loose-coupling, compositionality, and formal underpinning allow us to greatly enhance the design / management of a KDD process. Starting from a whole process, considering all the interactions among its tools and its users, it is possible to sketch out and identify coordination patterns that are typical in KDD processes. Among them, for instance, x-fold cross validation, parameter auto-tuning, choice of the best model after comparison of their accuracy, majority voting, parallel synchronized execution of different subprocesses, choice of first generated result after a parallel execution of different tools, optional preprocessing steps, and so forth. Once identified, such patterns can be extracted from the circuit and implemented as separated sub-circuits, each of which with its own interface and behavior. Since they have a formal semantics, each sub-circuit can be formally verified in order to guarantee that it complies with its intended behavior, and then reused as building blocks in many scenarios with different sets of tools and users. Moreover, editing the process for adding new steps or making new experiments is much simpler than using other languages, where interaction is not clearly distinguished from the rest of the processing. Aside from the aforementioned benefits, we recognize the need to address some issues that may arise in following the described approach. Such issues are not directly related to Reo, but arise from the peculiarities of the KDD support system that we are working on and the collaborative / distributed environment we refer to. Among them:

- data-awareness: in Reo some channels are data-aware (e.g., filter channel, transformation channel). This allows to define parametrized expressions in order to check, for instance, whether the current value flowing in a filter channel is less or greater than a threshold. However, in general Reo channels are not data-aware, so the designer have to explicitly introduce data-aware channels between two connectors in order to make the circuit fully data-aware. This may generate overly complex structures and make the design more difficult. Furthermore it would be desirable to add more semantic descriptions of data, e.g., to distinguish an accuracy parameter from a learning rate, both described by real values;

- explicit specification of behavior: the described approach relies on the explicit description of service’s behavior, in a specific formal model. Since the services we refer to are distributed and realized by different providers, we would need to ask each provider to add behavioral descriptions for their services, in order
use them in designing a process. Although this seems a strong limitation in
the application of the approach, we note that no real service integration can
be done without this information, and current research on process languages
are dealing with such an issue;

– coordination pattern design: Reo follows a non-conventional model for con-
currency, which is mostly new to programmers. Although learning how to
design a circuit is not a complex task, expertise is needed for realizing effi-
cient working circuits. This issue can be partially solved through a library of
reusable connectors for the most typical KDD patterns, as already explained;

Finally, note that the inherent complexity of model checking techniques was
observed also in this experimentation. We solved the problem by checking sepa-
ately each sub-circuit, and replacing (in the whole circuit) each of them with
simpler connectors with an equivalent behavior. This approach is feasible in
practice if a limited and stable number of connectors representing typical KDD
patterns can be defined. The analysis of this issue will be subject for future work.

References

2. Diamantini, C., Potena, D.: Semantic annotation and services for ldd tools sharing
and reuse. In: ICIDM Workshops, 1st Workshop on Semantic Aspects of Data
Mining, Pisa, Italy, IEEE (Dec 19 2008) 761–770
ACM SIGKDD Explorations 2(1) (June 2000) 24–28
(Jan. 2003) 89–93
Society (June 14–17 2005) 11–18
10. De Bourle, D., Coble, C., Stevens, R.: The Design and Realisation of the Vir-
tual Research Environment for Social Sharing of Workflows. Future Generation
with Data Assembly Lines. In: 4th ACM Workshop on Workflows in Support of
Large-Scale Science, Portland, OR, USA, ACM (Nov. 15 2009) 1–10