Contributions to a Semantically Based Intelligence Analysis Enterprise Workflow System

Robert C. Schrag, Jon Pastor, Chris Long, Eric Peterson, Mark Cornwell, Lance A. Forbes, and Stephen Cannon

***Abstract*—We have contributed key elements of a semantically based intelligence analysis enterprise workflow architecture: a uniformly accessible semantic store conforming to an enterprise-wide ontology; a branching context representation to organize workflow components' analytical hypotheses; a logic programming-based, forward-chaining query language for components to access data from the store; and a software toolkit embracing all the foregoing to streamline the process of introducing additional legacy software components as semantically interoperable workflow building blocks.**

We explain these contributions, focusing particularly on the toolkit. For certain widely used input/output formats—e.g., comma-separated value (CSV) files—a knowledgeable user can quickly "wrap" a newly installed component for workflow operation by providing a compact and entirely declarative specification that uses the query language to map specific relation arguments in the ontology to specific structural elements in the component's native input and output formats.

Our contributions are built to work with AllegroGraph, from Franz, Inc.

***Index Terms*—Intelligence analysis, enterprise workflow, hypothesis representation, branching contexts, semantic interoperability, declarative data transformation, software component wrapping**

I. Introduction

WE have contributed key elements of a semantically based intelligence analysis enterprise workflow architecture for Tangram, a multi-year, multi-contractor threat surveillance and alerting research and development program sponsored by the United States' Intelligence Advanced Research Projects Agency (IARPA). Tangram's objective has been to automate routine analysis workflows, so that these can be executed as standing processes, on a large scale.

Manuscript submitted August 19, 2009. This work was supported in part by the U.S. Government.

All authors were with Global InfoTek, Inc., 1920 Association Dr, Suite 600, Reston, VA USA 20191, 703-652-1600, (e-mail: firstinitialLastname@globalinfotek.com).

C. Long is now with SET Corp., Arlington, VA, 703-738-6214 (email: clong@setcorp.com).

L. A. Forbes is now with Solutions Made Simple, Inc., Reston, VA (email: lforbes@sms-fed.com).

To support the rapidly changing needs of an intelligence enterprise, a workflow authoring tool must be extremely flexible. The enterprise must be able to rearrange components (e.g., pattern matchers, classifiers, group detectors) in the same kind of way that a child rearranges Lego bricks. They must be able to introduce new software into the enterprise rapidly. However, Lego bricks have a distinct advantage over legacy software components from different source: they were all created to respect a common interface. One brute-force approach to integrating legacy components is to manually develop code that transforms data from one form (e.g., Java objects) to another (e.g., flat files); that requires $O(n^2)$ transforms. Tangram's approach reduces the required number of transforms to $O(n)$, and our toolkit enables knowledgeable users to "wrap" legacy components with such transforms, making the components workflow-ready quickly.

To motivate our contributions, we present the (notional, simplified) two-component workflow in Fig. 1: a suspicion scorer hypothesizes potential terrorists, then a group detector clusters the hypothesized terrorists into hypothesized potential terrorist groups.

Fig. 1 A notional intelligence analysis workflow

The workflow in Fig. 1 raises some enterprise-level architecture issues that our contributions address.

1) What are components' input and output data, how is data stored, and how do components access it? We have introduced a uniformly accessible semantic store conforming to an enterprise-wide ontology and a logic programming-based, forward-chaining query language for components to access data from the store. Component specifications (see Issue 3 below) indicate what data is accessed in particular.
2) How are the hypotheses that analytical components produce distinguished from background data, and how are they communicated among components? As hypotheses, analytical components' outputs must not simply be mixed indiscriminately with more uniformly credible evidence data or with each other. Among other considerations, the broad body of evidence changes over time (leading to different hypotheses), and different components—or

different (e.g., control) configurations thereof can lead to different hypotheses even for the same inputs. We organize the content of the semantic store into distinct RDF graphs that we call "datasets," and (correlating datasets with contexts) represent the outputs of successively applied analytical components as branching contexts (that incrementally add information). Our component specifications and our query language thus include parameters for the datasets that are passed among or otherwise accessed by components. Besides these datasets for hypotheses, the store includes one or more background, or "evidence," datasets and for convenience some intermediate (i.e., not necessarily hypothetical) datasets that result from purely logical queries. This treatment of evidence and hypotheses, together with the above-mentioned query language, provide a practical implemented solution to meet broad Tangram requirements outlined in [6].

3) How can legacy components with arbitrary input/output formats easily be made to interact with the data? The contributions above are integrated in a software toolkit to streamline the process of introducing additional legacy software components as semantically interoperable workflow building blocks. For certain widely used input/output formats—e.g., comma-separated value (CSV) files—a knowledgeable user can quickly wrap a newly installed component for workflow operation by providing a compact and entirely declarative specification that uses the query language to map specific relation arguments in the ontology to specific structural elements in the component's native input and output formats. The toolkit also provides some less fully automated interface options to address more general input/output situations.

II. Architectural Scheme of a Workflow Component

Fig. 2 presents our general scheme for wrapping legacy components.

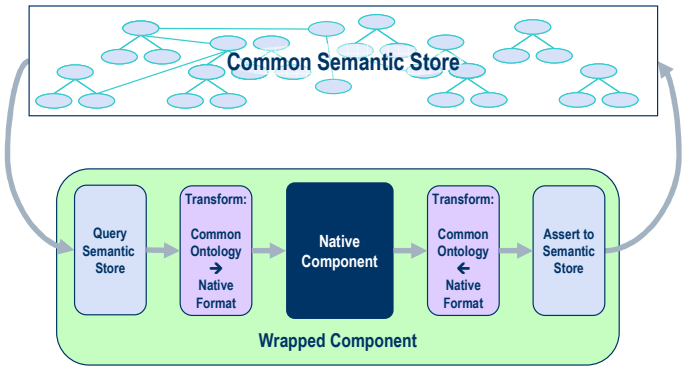

Fig. 2 Component wrapping scheme

Fig. 2 schematizes a single wrapped component that executes processes to:

1) Retrieve input data, expressed in the enterprise's common ontology, from the central semantic store.
2) Format the input data for the legacy component.
3) Invoke the legacy component in its "native" (unwrapped) form.
4) Convert the legacy component's native-format outputs to the common ontology, as metadata-bearing hypotheses.
5) Assert the output hypotheses to the central store.

We implement the central semantic store using AllegroGraph from Franz, Inc. AllegroGraph is a "quad" store that includes, in addition to the "subject," "predicate," and "object" fields standard to RDF and common to triple stores, a "graph" field. We use this field to distinguish among the various datasets that are available as inputs or have been produced as outputs of workflow components.

We provide a knowledge base (KB) query language supporting a wrapped component's query and assertion processes and allowing users to define, for specific analytical purposes, KB query components (including no legacy process) that combine elements from one or more existing datasets into one or more output datasets. We implement legacy component wrappers and KB query components using the Prolog and Common Lisp interfaces to AllegroGraph.

Fig. 3 illustrates the meta-data classes (noted in **bold**) and attributes (with multi-valued attributes starred*) that support the representation of a dataset's context lineage. We take each workflow component's execution, noted in a ProcessExecution (PE) object, as the source of the statements in any output (hypothesis) dataset; lineage is manifested in the connections among datasets, process executions, and workflow executions (noted in WorkflowExecution objects).

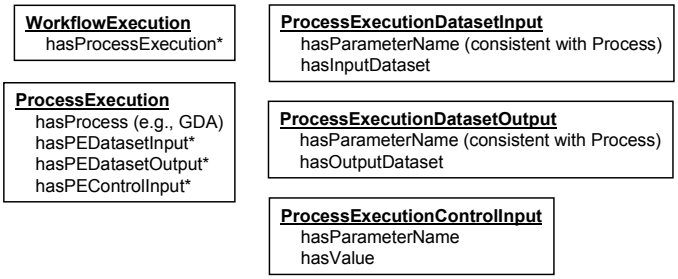

Fig. 3 Meta-data classes and attributes for hypothesis datasets

As noted in Section I, the interpretation of datasets as a context is incremental along its lineage: in general any statement that holds in a dataset that is upstream (workflow-wise) from a given dataset D created during a workflow also (implicitly) holds in D. The representation is thus space-efficient. We have not yet found it necessary to implement such transitivity of dataset contexts directly in the KB query language; our current workflow components use just background (evidence) datasets and datasets that their immediate workflow predecessors create.

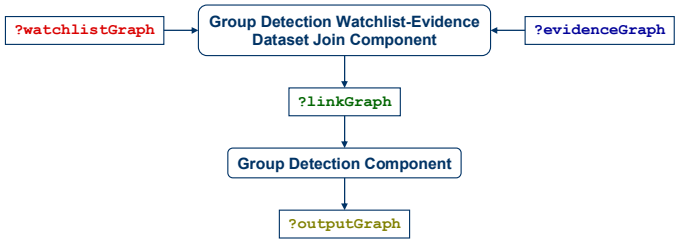

Fig. 4 Use case workflow (see Section III)

III. Use Case Workflow

Fig. 4 presents a use case workflow including both a wrapped legacy component and a KB query component.

In Fig. 4, datasets (graphs) are depicted by square-cornered boxes; workflow components are depicted by round-cornered boxes. Each component reads data from one or more input graphs and writes to one or more output graphs. Here, a dataset join KB query component is used to select from broader evidence (right) just information relevant to watchlisted terrorist suspects (left) for processing by a downstream legacy group detection component.

In our toolkit, the defining forms for workflow components are Lisp macro calls. Beyond providing one or more files containing such definitions, ToolKit users need never interact directly with Lisp or with AllegroGraph, as we provide alternative interfaces.

IV. KB Query Components and Query Language

The definition for the KB query component used in Fig. 4 appears below.

```
(defKB-query-component
    group-detection-watchlist-evidence-dataset-join-component
  ((and (q- ?Event !rdf:type !teo:TwoWayCommunicationEvent
            evidenceGraph)
        (q- ?Event !teo:sender ?sender ?evidenceGraph)
        (q- ?Event !teo:receiver ?receiver ?evidenceGraph)
        (q- ?sender !rdf:type !teo:Person ?evidenceGraph)
        (q- ?receiver !rdf:type !teo:Person ?evidenceGraph)
        (q- ?sender !rdf:type !teo:Person ?watchlistGraph)
        (q- ?receiver !rdf:type !teo:Person ?watchlistGraph)
        (a- ?Event !rdf:type !teo:TwoWayCommunicationEvent
            ?linkGraph)
        (a- ?Event !teo:deliberateActor ?sender ?linkGraph)
        (a- ?Event !teo:deliberateActor ?receiver ?linkGraph)
        (a-- ?sender !rdf:type !teo:Person ?linkGraph)
        (a-- ?receiver !rdf:type !teo:Person ?linkGraph))))
```

The above component selects events from one dataset (denoted by the logic variable ?evidenceGraph) whose participants also appear in another dataset (denoted by ?watchlistGraph) and asserts the links among them in an output dataset (represented by the logic variable ?linkGraph) for consumption by a group detection component. Note the following.

- This component performs a single KB query that implicitly conjoins (logically) the twelve top-level (q-, a-, and a--) forms.
- A q- conjunct succeeds iff a triple (in subject, predicate, object, graph, index—"spogi"—format) exists in the workflow KB. q- is included in the standard Franz Allegro Prolog interface to AllegroGraph.
- a- indicates that a triple is to be written to the specified output dataset. An a- conjunct always succeeds. a- and its duplicate-avoiding twin a-- (below) are our contributions that confer the KB query language's forward chaining character.
- a-- indicates that a triple is to be written to the workflow KB iff it is not already present there. An a-- conjunct always succeeds.
- !rdf:type is an example of a shorthand that expands to http://www.w3.org/1999/02/22-rdf-syntax-ns#type — the atom type in the namespace for RDF. (!teo: refers to an application-specific ontology.)
- ?Event, ?sender, and other symbols beginning with ? are logic programming (AKA Prolog) variables. In the logic programming style we support, every logic variable becomes bound when the q- conjunct is matched in the KB.
- Prolog will backtrack to execute each conjunct in the KB query for every combination of variable bindings for which the preceding conjuncts succeed.
- The KB query language provides a variety of additional constructs (e.g., and, or, not) in which the usual expressions that appear as top-level conjuncts may be embedded—e.g.,

```
(and (not (q- ?P !rdf:type !teo:Terrorist ?evidenceGraph))
     (or (q- ?P1 !rdf:type !teo:Terrorist ?evidenceGraph)
         (q- ?P2 !rdf:type !teo:Terrorist ?evidenceGraph))).
```

- While the repetition of entity type statements—e.g.,

```
(a-- ?sender !rdf:type !teo:Person ?linkGraph)
```

—from the input graph is not strictly necessary given our context interpretation, the Tangram contractors agreed that it would be convenient to include such declarations uniformly in all datasets.

Below are the definitions for some utility KB query components that we provide with the toolkit distribution.

```
(defKB-query-component 2-input-dataset-union-component
    (DataUnionProcess)
  ((query (q- ?S ?P ?O ?sourceGraph1)
          (a- ?S ?P ?O ?destGraph))
   (query (q- ?S ?P ?O ?sourceGraph2)
          (a- ?S ?P ?O ?destGraph))))

(defKB-query-component 3-input-dataset-intersection-component
    (DataIntersectionProcess)
  ((query (q- ?S ?P ?O ?sourceGraph1)
          (q- ?S ?P ?O ?sourceGraph2)
          (q- ?S ?P ?O ?sourceGraph3)
          (a- ?S ?P ?O ?destGraph))))

(defKB-query-component dataset-de-duplication-component ()
  ((query (q-  ?S ?P ?O ?sourceGraph)
          (a-- ?S ?P ?O ?destGraph))))
```

The (first) dataset union component writes everything it finds in either of its source graphs into its destination graph; the (second) intersection component writes anything it finds in

all of its sources into the destination. A workflow author may choose to follow either of these up with the (third) dataset de-duplication component to remove duplicates; note that the author could achieve the same effect by using a-- rather than a-conjuncts in the union components' definitions.

Existing Tangram workflow and process infrastructure required that we specify the fixed (e.g., two-input) arities for the components above. This might not be the case in every workflow setting of interest (see Section VIII). Likewise, it might not be necessary to name (or permanently componentize) every query before it can be used.

V. Wrapped Legacy Components

Toolkit users define wrappers for legacy/native components using the Lisp macro defWrapped-component, which affords a choice among three distinct interfaces. Non-Lisp-programming ToolKit users will want to use one of the first two interfaces described below; Lisp-programming users are most likely to use the first or third.

1) Fully automatic: defWrapped-component writes a comma-separated value (CSV) or other delimited text file (to be consumed by the native component) for each input dataset and automatically reads a delimited text file (produced by the native component) for each output dataset. For native components with delimited text file-oriented input/output, the ToolKit user need provide no additional wrapping code.
2) Semi-automatic: defWrapped-component automatically writes an ntriples file for each input dataset and automatically reads an ntriples file for each output dataset. The ToolKit user provides additional (presumably non-Lisp), shell-callable wrapping code as necessary to mediate between these ntriples files and the native component.
3) Manual: The ToolKit user provides, via an additional argument to defWrapped-component, custom Lisp code to implement the required native component interface. Here we assume that the Lisp programmer will interact directly with AllegroGraph to create suitable inputs for the native component.

In the sequel, we focus primarily on the fully automatic interface.

Consider the GDA group detection algorithm [3] from CMU's Auton Lab), which uses CSV input and output files as shown in Fig. 5. The group detector uses event-based linkages among individuals to infer groups of associating individuals. Each input line indicates evidence that a certain event involves a certain individual. Each output line indicates that a certain individual is hypothesized to belong to a certain group.

Native GDA Input:

```
Ev-1194,In-10381
Ev-709,In-15840
Ev-709,In-36232
Ev-38749,In-4938
Ev-38749,In-48834
Ev-34121,In-3007
Ev-34121,In-35214
Ev-65474,In-21371
Ev-65474,In-19354
Ev-23484,In-39017
Ev-23484,In-16809
…
```

Native GDA Output:

```
group,entity
G0,In-10096
G0,In-15840
G0,In-19354
G0,In-19540
G0,In-19625
G0,In-21371
G0,In-28719
G0,In-37201
G0,In-37733
G0,In-38634
G0,In-47910
G1,In-1002
…
```

Fig. 5 CSV input/output files for the GDA group detection component

Below is a toolkit-based component definition that invokes the automatic CSV file interface to wrap GDA. The (completely declarative) definition specifies that GDA-component-TerroristGroup is an instance of the class GroupDetectionProcess (see [9]). The (keyword) argument :native-input-CSV-file-specs specifies the relation of the input CSV file (to be named "GDA-input-links.csv") to the input dataset (bound to the Prolog variable ?linkGraph).[1] Note that the separating character may be specified, using the :text-delimiter argument, and the presence of a headerline via the :headerline argument. The argument :native-output-CSV-file-specs specifies the relation of the output CSV file (to be named "GDA-output-groups.csv") to the output dataset (bound to ?outputGraph). The remaining top-level arguments specify how to invoke the native component. Further explanation follows the definition.

```
(defWrapped-component GDA-component-TerroristGroup
   (GroupDetectionProcess)
  :native-input-CSV-file-specs
   (("GDA-input-links.csv"
     :query
      (query
       (q- ?E !teo:deliberateActor ?P ?linkGraph))
     :query-type select
     :headerline nil
     :text-delimiter ","
     :query-template (?E ?P)))
  :native-output-CSV-file-specs
   (("GDA-output-groups.csv"
     :query
      (query
       (a- ?G !teo:orgMember ?P ?outputGraph)
       (a-- ?G !rdf:type !teo:TerroristGroup ?outputGraph)
       (a-- ?P !rdf:type !teo:Terrorist ?outputGraph))
     :headerline t
     :CSV-template (?G ?P)
     :namespace-template
      ("http://anchor/teo#" "http://anchor/teo#")))
  :native-component-directory "GDA_DISTRIBUTION"
  :native-component-command-name "gda_applic"
  :native-component-command-arguments
   ("GDA-output-groups.csv" "GDA-input-links.csv"))
```

[1] The full interface supports any number of native input and of native output delimited text files and corresponding datasets/graphs.

Fig. 6 illustrates how the :native-input-CSV-file-specs argument is processed.

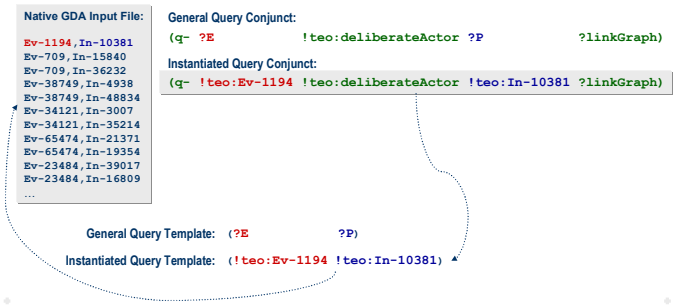

Fig. 6 Automatic CSV file input mechanism

First, we execute the input query against the input dataset (graph). At top right, Fig. 6 illustrates how the query's single (general) conjunct is first specifically instantiated, binding the conjunct's variables to values for which a triple exists in the input graph. The :query-template argument specifies how the query's bound variable values should be ordered in the CSV file. At bottom, Fig. 6 illustrates the intermediate step of instantiating the query template, based on the instantiated query conjunct. At left, Fig. 6 shows how we generate one CSV file line per query instantiation.[2] (Note that the RDF namespace, !teo:, is removed, as it is not useful to the native component.)

Fig. 7 illustrates how the native component is (next) invoked by the workflow execution system. Execution takes place in a temporary directory specific to the given workflow and component instance.

Fig. 7 Automatic CSV file native component calling mechanism

Fig. 8 illustrates how the :native-output-CSV-file-specs argument is (next) processed.

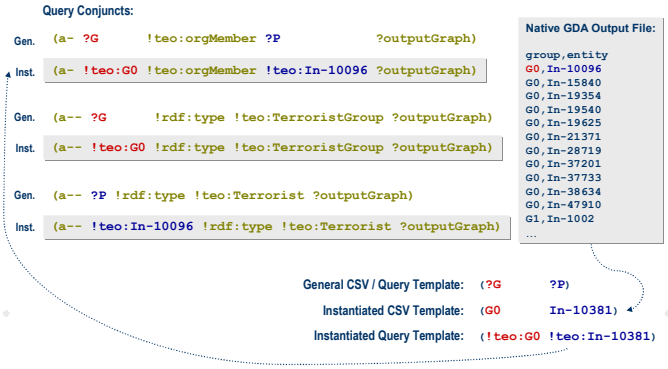

Fig. 8 Automatic CSV file output mechanism

The process is here roughly the reverse of that in Fig. 6. At bottom, Fig. 8 illustrates how we first interpret each line of the output CSV file (at right) using the template specified (via the :CSV-template argument), instantiating the template and binding query variables. Again, the template indicates the order of each bound Prolog variable in each line of the CSV file. Note the final template instantiation step that inserts appropriate RDF namespaces (per the :namespace-template argument). At right, Fig. 8 illustrates how these bindings are used to instantiate each specified output assertion (query conjunct). Each assertion is executed to add a triple to the semantic store (with appropriate treatment of duplicates).

VI. Conceived Full Automation for Components with XML Input/Output Files

While delimited text input/output formats are quite prevalent, they are by no means the only structured formats of interest. We have also designed (not yet implemented) a similar, declaratively-specified wrapping capability for components with XML file input/output. The general idea is to embed a similar query specification into the XML file where data is to be read or written. Another alternative on the input side (only) would be integration of Xpath and Xquery with logic programming. (See [1] for a recent survey.)

VII. The Wrapping Process

The toolkit's comprehensive documentation (available from the first author) details the following steps included in the end-to-end process of wrapping and then deploying components.

1) Install the wrapping toolkit.
2) Install the native component so that it will be accessible to the wrapper.
3) Define any KB query component(s) needed to select appropriate data from any broader dataset(s).
4) Define the wrapper for the native component.
5) Test both KB query and wrapped native components to ensure effective operation. We have developed and applied a testing framework that includes component concurrency (i.e., re-entrance) testing.
6) Deploy the developed and tested components.

These steps may of course be undertaken by different classes of users. E.g., in a component wrapping team (of which an enterprise may have several), one member (the "installer") may be primarily responsible for software installations; another (the "developer") may be expert with the enterprise's ontology, workflows, and datasets, the KB query language, and the component defining forms; still another (the "tester") may primarily have testing and another (perhaps the "installer" again) deployment responsibilities. "Scripters" might write custom Lisp wrapping code or shell scripts or other command line-callable programs to perform data transformations not (yet) supported by toolkit (semi-) automation.

For each component to be wrapped, the wrapping team also should include, or at least have access to, a component "champion" who knows what enterprise function(s) the component must accomplish and understands how the component works well enough to address any wrapping issues

[2] This is per the value **`select`** specified for the **`:query-type`** argument, which indicates that duplicate links (useful to GDA) are to be retained in the input dataset. By instead using the (default) value **`select-distinct`**, the user may alternatively specify one line per unique query instantiation (thus removing duplicates).

(e.g., whether duplicate assertions are or are not appropriate, what native component control parameters are appropriate). The champion should bring one or more exemplary use cases (preferably expressed in terms of the enterprise's datasets and ontology) and should help the wrapping team realize the use case(s) in component (and workflow) definitions.[3]

Finally, the component wrapping team always should be able to present new requirements to the toolkit development team (who may serve multiple enterprises).

We developed the toolkit during roughly six months of concentrated effort, to serve both the broader Tangram community and ourselves. Starting with the use case presented in Section III, we developed first the KB query language and KB query components, then progressively more automatic interfaces with which we wrapped GDA (initially). We also have used (or assisted others to use) the toolkit to wrap the ORA group detection algorithm, suspicion scorers based on the Proximity [7] and NetKit [5] classifiers, and the pattern matchers LAW [9] and CADRE [8].

We have met the Tangram program's toolkit usability goals: as knowledgeable users, we can usually (for components with inputs/outputs amenable to the toolkit's fully automatic interface) complete Steps 3 and 4 of the above wrapping process within a single staff hour.

VIII. Relaxing the Context Monotonicity Assumption

Implicit in the semantics of current Tangram workflow processing is the following monotonicity assumption: A component's output graph(s) only add(s), logically, to the information in its input graph(s), never delete(s) or retract(s). This is not entirely practical.

The need to manage potentially conflicting source information and analytic hypotheses is ubiquitous in an intelligence analysis enterprise. An analyst, surrounded with data and applicable tools or methods, may choose to pursue one line of reasoning at one time and another later, and different analysts may take different approaches and may build on each other's analyses or workflow products. Each such approach—a combination of data, tools, methods, and earlier hypotheses—represents a context for analytical reasoning. It is important within the enterprise for each analyst to understand the actual context of each piece of information that s/he might examine and exploit in further analysis—in which s/he may either extend an existing context or branch to create a new subcontext.

Different contexts may arise in workflow-supported analytical reasoning for different reasons, including:

- Differences in supporting data, from:
  - Conflicting original data sources.
  - Time-varying data conditions for a given source, such as:
    - Disbelief in something we earlier had belief in (perhaps because it had been supplied in error).
    - Belief in something we did not have belief in (perhaps because we had no data about it).
- Differences in supporting analytical hypotheses, from:
  - Analyst's conjecture, or "what-if" analysis (that may effect belief or disbelief in data as discussed above).
  - Differences in workflow components giving rise to different answers, when:
    - A given workflow function has alternative realizations in different components.
    - A given component has alternative configurations of control parameters.

We have commenced efforts to address these issues both formally and with appropriate workflow system infrastructure.

IX. Contributions' Relevance Beyond Tangram

The use case workflow in Section III includes a generic "Group Detection Component." While we've noted (in Section V) that GDA-component-TerroristGroup is an instance of the class GroupDetectionProcess, we haven't said anything yet about how such a specific component instance is selected from among the available alternatives for such a general process class. Beyond enabling semantic interoperability of enterprise workflow components, IARPA's broader objectives in Tangram have included providing technology for characterizing, for a given generic workflow process, the likely performance of a given specific component with data inputs having certain characteristics, so that the workflow management system can select the component likely to perform best in any given circumstance. Our toolkit supports this objective by automating the formal description and registration of newly defined components in Tangram's process catalog [9].

It's worth noting that all of the toolkit's other heretofore-described capabilities remain applicable in the (perhaps more pragmatic) setting where users specify particular components for all workflows themselves.

[3] Consider that a champion may also bring a new data source that may require extensions or other modifications to the enterprise ontology. Addressing such issues has been the responsibility of a different Tangram contractor.

References

[1] Almendros-Jiménez, J. M., Becerra-Terón, A., Enciso-Baños, F. J.: Querying XML documents in logic programming, Theory Pract. Log. Program. 8, 3 (May. 2008), 323–361.

[2] Carley, K. M., Dereno, M.: ORA—Organizational Risk Analyzer. Tech. rep. CMU-ISRI-06-113, Carnegie Mellon University, August 2006.

[3] Kubica, J.; Moore, A.; Schneider, J., Tractable group detection on large link data sets, Third IEEE International Conference on Data Mining (ICDM-2003), pp. 573–576, 19–22 Nov. 2003

[4] Macskassy, S. A., Provost, F.: NetKit-SRL: A Toolkit for Network Learning and Inference, In Proceedings of the NAACSOS Conference, June 2005.

[5] Murray, K., Harrison, I., Lowrance, J., Rodriguez, A., Thomere, J., Wolverton, M.: PHERL: an Emerging Representation Language for Patterns, Hypotheses, and Evidence, in Proceedings of the AAAI Workshop on Link Analysis, 2005.

[6] Neville, J., Jensen, D.: Dependency networks for relational data. In Proceedings of the 4th IEEE International Conference on Data Mining, 2004.

[7] Pioch, N.; Hunter, D.; Fournelle, C.; Washburn, B.; Moore, K.; Jones, E.; Bostwick, D.; Kao, A.; Graham, S.; Allen, T.; Dunn, M.: CADRE:

continuous analysis and discovery from relational evidence, International Conference on Integration of Knowledge Intensive Multi-Agent Systems, 2003. pp. 555–561, 30 Sept.–4 Oct. 2003.

[8] Wolverton, M., Berry, P., Harrison, I., Lowrance, J., Morley, D., Rodriguez, A., Ruspini, E., Thomere, J.: LAW: A Workbench for Approximate Pattern Matching in Relational Data. In Proceedings of the Fifteenth Innovative Applications of Artificial Intelligence Conference (IAAI-03), 2003.

[9] Wolverton, M., Martin, D., Harrison, I., Thomere, J.: A Process Catalog for Workflow Generation, in The Semantic Web—7th International Semantic Web Conference, Springer, vol. 5318/2008, pp. 833–846, 2008.