AFRL-IF-RS-TR-2006-232
Final Technical Report
July 2006

FUSEFLOW: A WORKFLOW-AWARE FUSELET MANAGEMENT ENVIRONMENT

Architecture Technology Corporation

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-232 has been reviewed and is approved for publication.

APPROVED: /s/

DALE W. RICHARDS
Project Engineer

FOR THE DIRECTOR: /s/

JAMES W. CUSACK
Chief, Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2006	Final	Mar 05 – Mar 06

4. TITLE AND SUBTITLE
FUSEFLOW: A WORKFLOW-AWARE FUSELET MANAGEMENT ENVIRONMENT

5a. CONTRACT NUMBER
FA8750-05-C-0084

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Matthew Stillerman, Robert Joyce, Andrew Chruscicki, Brady Tsurutani

5d. PROJECT NUMBER
558J

5e. TASK NUMBER
TD

5f. WORK UNIT NUMBER
03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Architecture Technology Corporation
33 Thornwood Drive, Suite 500
Ithaca New York 14850

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFSE
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-232

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 06-494

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Information spaces deployed under the Joint Battlespace Infosphere (JBI) program utilize small transformation – fuselets – to build up applications of meaning to end users. FuseFlow is a design, management, and performance monitoring system for structured collections of JBI fuselets and non–fuselet clients. These collections, termed fuselet workflows, have clear meaning to end users and address significant challenges for fuselets in the areas of scale, concurrency, authoring, and maintenance. In addition, fuselet workflows begin to capture author intentions about how fuselets will interact and how end-user goals will be achieved; “meaning” is inherent in the collection, at a scale above individual fuselets. The FuseFlow system is founded on well-known techniques for designing business processes (workflows) to achieve customer satisfaction and for continuously improving the process. At the same time, FuseFlow components interact with existing JBI structures using established APIs, in order to implement the Fuseflow fuselet workflow abstraction.

15. SUBJECT TERMS
JBI, Fuselets, Workflow, XML

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dale W. Richards
U	U	U	UL	50	19b. TELEPONE NUMBER (*Include area code*)

Table of Contents

1. Summary 1
2. Introduction 5
   2.1 Background 5
      2.1.1 JBI and Fuselets 5
      2.1.2 Workflows 5
   2.2 FuseFlow Project Motivation 6
   2.3 Scope of the FuseFlow Project 7
3. Methods, Assumptions, and Procedures 8
   3.1 FuseFlow Architecture 8
      3.1.1 FuseFlow Authoring and Compiling 8
      3.1.2 FuseFlow Process Registration, Monitoring, and Control 10
   3.2 FuseFlow Language 13
      3.2.1 Process Definition 13
      3.2.2 Control-Flow Logic 13
      3.2.3 Declarative Structure 17
      3.2.4 Data Fields 18
      3.2.5 Tests 19
      3.2.6 Error Handling 19
      3.2.7 External Context 20
      3.2.8 Performance Dimensions 20
      3.2.9 Formal Parameters 21
      3.2.10 Activity Specifications 21
   3.3 FuseFlow Authoring Tool 21
   3.4 Process Enactment 25
      3.4.1 Tasks and the Central Control Fuselet 25
      3.4.2 Alternative Enactment Architectures 28
      3.4.3 FuseFlow Compiler Prototype 29
   3.5 Process Assessment and Improvement 30
      3.5.1 Integrated Support for Assessment and Improvement 31
4. Results and Discussion 34
5. Conclusions 36
6. References 38

Appendix A: Notes on Information Space Problems, Diagnosis, and Remediation 39
A.1 Missing Objects 39
   A.1.1 Diagnosis – Missing Objects 39
A.2 Inaccurate Objects 40
   A.2.1 Diagnosis – Inaccurate Objects 40
   A.2.2 Remedies – Inaccurate Objects 41
A.3 Extraneous Objects 41
   A.3.1 Diagnosis – extraneous objects 41
   A.3.2 Remedies – extraneous objects 42
A.4 Tardy Objects 42
   A.4.1 Diagnosis – Tardy objects 42
   A.4.2 Remedies – Tardy objects 43

A.5 Usability Problems 43
A.6 Process Too Expensive 43
A.6.1 Remedies – process too expensive 44

Lists of Figures

Figure 1. FuseFlow System Architecture... 9

Figure 2. Monitoring and Control Client user interface 11

Figure 3. Example workflow diagram 13

Figure 4. Activities with multiple transitions in or out: splits and joins......................... 14

Figure 5. Graphical notation to distinguish AND and XOR control flow. 14

Figure 6. Using a Route in a workflow to express complex control flow. 16

Figure 7. FuseFlow Process Editor 22

Figure 8. Setting the properties of an activity.. 23

Figure 9. FuseFlow Runtime Architecture.. 25

Figure 10. Data flow between workflow fuselets 27

1. Summary

The Joint Battlespace Infosphere (JBI), a research effort of the Air Force Research Laboratory, addresses the challenge of presenting a shared space of high-quality managed information to all concerned warfighters. Information transformation is one of the core capabilities of the JBI infrastructure, converting information that is already present to a form that will be useful to end users and their applications. Fuselets are small autonomous programs that perform information transformations as part of the JBI system. It is anticipated that a typical JBI information space may have many running fuselets, and that transformations of interest will arise when fuselets act together on the information. Thus arises the problem of how to orchestrate the collaboration of groups of fuselets to achieve a desired result. The research effort described in this report investigates one approach to such collaboration – to impose a workflow-like structure on the interacting fuselets.

The JBI information space model consists of particles of information called *managed information objects (MIOs)*. Each MIO consists of some *metadata*, and possibly a *payload*. Clients of the JBI platform (computer programs) can *publish* objects into the information space, they can receive new objects by *subscription*, and they can *query* for information objects that have been previously published. Object publishers supply part of the metadata that they are publishing – the platform supplies the rest. Subscribers and those issuing queries can specify a predicate that constrains the objects that they will receive.

A fuselet is an autonomous JBI client. It receives information objects by subscription and/or query, and it publishes transformed information objects. If some other fuselet receives this newly published object via subscription, then there will be a flow of control and data from the first fuselet to the second. Suppose a whole chain of such linked fuselets (or a tangle) is needed to effect the required transformation. How can this fuselet collaboration be conveniently arranged to take place? How can its functioning be meaningfully monitored and managed? For an arbitrary set of interacting fuselets this seems daunting, if not impossible. A range of pathologies such as cycles, race conditions, and deadlock are possible. The sheer number of fuselets may make management impractical. Amidst this chaos, an individual fuselet may not be performing a recognizably useful task, in spite of being a required part of the overall transform.

One of the themes of the JBI program is that information spaces will be managed by warfighters who are not necessarily expert programmers. Information management staff will deploy fuselets and may author some of them in order to tailor the information space to the evolving situation in the theater of war. Clearly, problems of complexity and concurrency are seriously at odds with this vision.

The flow of data and control between fuselets, described above, resembles superficially what happens in a conventional business workflow – when one task is completed, the data and control is passed to some other actor for the next task. The term “flow of control” means that the second task cannot start until the first one is completed, as well as determining what that second task is and who will perform it. Of course, enterprise workflow management systems provide powerful

tools for constructing, visualizing, and controlling such linked tasks. The approach taken for managing large collections of interacting fuselets is to exploit this analogy. Similar software tools and infrastructure were considered for use in constructing, visualizing, and controlling *fuselet workflows*.

Modern enterprises do not just implement their business processes as workflows and then ignore them as they run. Rather, they carefully monitor and evaluate the performance of business processes against internal and external measures. Using this data, they try to identify opportunities for process improvement. Implementing business processes using a workflow management system supports this style of performance assessment and improvement in two ways: First, it provides a clear, uniform view of how the process actually runs. Such systems typically can generate a detailed log of all activity, suitable for off-line analysis. Second, the workflow management system makes it very easy to rearrange the process to implement improvements.

An additional goal of this research effort was to study how performance assessment and improvement techniques can be adapted from the world of business process and applied to fuselet workflows, thus extending the analogy.

The effort started with a high-level architecture for an integrated system called FuseFlow that will support the entire lifecycle of fuselet workflows: design and authoring, compiling, instantiation (loading), enactment, monitoring, performance assessment, and improvement. The FuseFlow architecture (See Figure 1. FuseFlow System Architecture on page 9.) shows the FuseFlow system layered on top of the JBI. The envisioned system is shown coupled with iFUSE, a fuselet authoring and analysis tool that is currently under development at ATC-NY.

A key feature of the system is the FuseFlow language, a process definition language with some additional features related to information transformation and to process improvement. This language defines what fuselet workflows are, abstractly. The purpose of the FuseFlow system is then to enable authoring of specifications in this language, to map these abstract specifications into running fuselets, and to use the specification as a guide in evaluating the performance of the running fuselet workflow. The limits imposed by the FuseFlow language are important – by restricting the ways in which fuselets can interact with one another, clarity is increased, some difficult concurrency problems are avoided, and management is made tractable.

The FuseFlow language was designed during the course of this effort. The system architecture was refined, and early prototypes of its major components wree implemented. The result is not fully integrated, and the components have somewhat restricted functionality. This software, capable of running on ISX's FREeME fuselet execution environment, was delivered together with an end-to-end demonstration of authoring, compiling, loading, enactment, monitoring, and control of a fuselet workflow.

The FuseFlow language is a dialect of XML. It borrows some syntax and concepts from XPDL[3], a process definition language standardized by the Workflow Management

Coalition[4]. Fuselet workflows are specialized in that their purpose is information transformation. To support this, the FuseFlow language has syntax for declaring the "signatures" of workflows and of individual activities – declarations of the information objects that will be input or output.

The processing in FuseFlow (as with other workflow systems) is divided into "cases". Each case is triggered by the publication of one or more information objects, and may result in the publication of some other objects. The model for the performance of a fuselet workflow is that each case may be considered to be OK or defective. It is the rate and seriousness of defects (as perceived by stakeholders) that constitutes performance. The nature of potential defects in a workflow's cases will be domain-specific and possibly specific to that workflow's implementation. Thus, the workflow author must specify the defect modes that are possible, called *performance dimensions*, and the means of measuring each one.

Compiling a FuseFlow specification yields a detailed list of the fuselets that should be run (i.e. fuselet classes and corresponding sets of parameters). One of the major project results is a method of robustly enacting workflows with fuselets. Each task (or "activity") of the workflow is carried out by a separate fuselet called a *Task-Bearing Fuselet* (TBF). The TBFs contain code to perform their respective tasks. They communicate with a *Central Control Fuselet (CCF)* by means of publication and subscription to certain control objects. The CCF orchestrates the TBFs, telling each one, in its proper turn, when to run, and which data to act on. A separate *Trigger Fuselet* detects the conditions for a new case, and signals the CCF.

This runtime picture is completed by a *Monitoring and Control Client* which handles workflow instantiation, starting and stopping, and progress monitoring. It also subscribes to the control objects to learn the status of workflow cases. It uses the Fuselet Management API to directly load, start, and stop the fuselets.

The FuseFlow authoring component, called the *Process Editor*, is a graphical editor which uses a standard workflow diagram notation as its central interface metaphor. This component is implemented as an Eclipse plug-in. It shares some code with iFUSE (also implemented as Eclipse plug-ins) and is thereby weakly integrated with it. See Figure 7 for a screen shot from this editor.

The work on performance assessment and improvement is primarily conceptual. Aspects considered during the effort include: how the improvement cycle might work, practical barriers to carrying it out, and integrated support within FuseFlow that would likely be beneficial. However, these tools have not been designed in detail, or built. *Generic categories of deficiencies* in information spaces were analyzed. Such deficiencies were broke down into six exhaustive categories. For each form of deficiency strategies for diagnosis and remediation were suggested. The minimal can be found in Appendix A of this report.

At this stage the major conceptual work on FuseFlow has been completed. The prototype is poised to be integrated and polished –no major technical difficulties are foreseen in completing

the vision of an integrated FuseFlow system. Such a system could be applied to real-world problems.

The effort has provided insight into many subtle, unanticipated issues in fuselet workflows (and information management, more generally). Further work will be able to address these problems, build a polished, integrated system, and characterize its performance and effectiveness in a realistic environment.

2. Introduction

This report describes the results of the FuseFlow project, a research effort of ATC-NY in collaboration with ISX Corporation and SBS, Inc., funded by AFRL/IF. FuseFlow imposes a workflow-like structure on collections of autonomous transformations ("fuselets") and client programs within an information system, such as the Air Force Research Laboratory's Joint Battlespace Infosphere (JBI). Each fuselet or client corresponds roughly to a workflow task. This structure, which resembles a conventional business workflow, is imposed on fuselets in order to make them easier to manage and to address problems of concurrency. A fuselet workflow is a way of structuring a collection of fuselets that will interact with one another via the JBI for some purpose.

2.1 Background

2.1.1 JBI and Fuselets

The Joint Battlespace Infosphere (JBI) is a research program [1] of the Air Force's Research Laboratory, Information Directorate (AFRL/IF) leading towards a new breed of middleware for military information systems. This program has produced a series of middleware specifications and prototypes collectively referred to as "the JBI."

Information in the JBI consists of a shared pool of *Information Objects (IOs)*. Each object is described by XML metadata and may also have a payload. Clients are programs that connect to the JBI and access the pool of IOs. Clients may *publish* new IOs, and may receive objects whose metadata matches a given predicate. With a *query*, clients receive requested objects synchronously. In contrast, a *subscription* causes new matching objects to be delivered to the client as they are published.

Fuselets [2] are autonomous JBI clients (i.e. running without human intervention) that are part of the JBI platform. They function solely by publishing, subscribing, and querying IOs (they have another interface for management purposes). Fuselets are intended to carry out simple transformations on the information in the infosphere, thereby increasing the usefulness of that information to end users.

2.1.2 Workflows

Workflows are the means by which organizations automate and coordinate the distribution of work in order to achieve their business purposes. This is quite a mature subject in business management circles, with lots of academic work as well as many software systems and tools, both commercial and free.

In this section, a few key points that will be relevant later in the report are presented, and the reader is directed to some sources of in-depth information.

Workflow management systems (WfMSs) orchestrate the tasks that are to be performed, who (or what) will perform them, and in what order. The actual work (task performance) generally lies outside the scope of the WfMS.

A WfMS follows a process definition (a piece of data) as it steps the participants through the process. The process definition may easily be edited to effect a change in the business process. One of the chief advantages of WfMSs as conventionally employed is the enhanced flexibility implied by this structure, as compared with a "hard-coded" process.

The processes directed by a WfMS are repetitive. The process definition is, in effect, a template for work to be done each time certain conditions are met. This event is referred to as a *trigger* and the resulting work as a *case*. Thus, in response to each trigger, the WfMS orchestrates a cascade of tasks that are all part of the same case. Multiple cases can be in progress at the same time, and will proceed independently. The process definition may highly constrain how the tasks of a case are executed in relationship to one another. However, such specifications will not constrain how tasks of distinct cases are related.

For in-depth information about workflow, perhaps the most insightful place to start is the Workflow Management Coalition (WfMC) whose definitive Workflow Reference Model published in 1993 has withstood the test of time. Their other published works include a standard process definition language called XPDL and a wide range of articles about workflow.

The Organization for the Advancement of Structured Information Standards (OASIS) has an ongoing effort to standardize a workflow language called WSBPEL (or Web Services Business Process Execution Language). The committee writing this standard is a who's-who of large business software companies.

There are many books and journal articles on workflow, in all of its aspects. Two that were found helpful were: *Workflow Management: Models, Methods, and Systems*, by Wil van der Aalst and Kees van Hee[10], and *Design and Control of Workflow Processes* by Hajo Reijers[9].

2.2 FuseFlow Project Motivation

Running instances of the JBI are expected to employ many fuselets in order to tailor the information that they contain for specific purposes. Naïvely, this situation would pose several challenges that FuseFlow is intended to address:

- A large number of fuselets may be difficult to manage: author, deploy, monitor, control, repair, and optimize. Individual fuselets may perform small transformations whose place in a larger-scale purpose is difficult to discern unless it is well documented.

- Fuselet inputs will, in general, arrive asynchronously. Writing and deploying fuselets to handle this asynchrony, especially when they must interact with one another, can be difficult.

- Fuselets are intended as a tool for information management – the design and operation of information spaces to meet specific goals. How can managers know whether those goals are being met? How can they modify the way fuselets are deployed to meet the needs of end users better?

FuseFlow imposes a workflow-like structure on collections of fuselets. Each fuselet corresponds roughly to a workflow task. When triggering conditions are met, a new "case" of the workflow is created, and starts executing. A specification written in the FuseFlow language defines the passage of control between fuselets as the case is handled. As each task is completed, the fuselet publishes IOs that enable the "downstream" tasks for that case to start running in other fuselets. The design calls for a *central control fuselet (CCF)* that orchestrates the flow of control between fuselets, for each case.

Another important aspect of the FuseFlow project is process monitoring and improvement. Just as with conventional workflows, one of the main motivations for adopting fuselet workflows is that they enable systematic monitoring of processes. Process monitoring can then be used to assess the performance of the process. Such assessments can, in turn, drive process improvement. Workflow automation makes improvements easier to institute, due to looser coupling between tasks.

2.3 Scope of the FuseFlow Project

The work reported on here was carried out under contract FA8750-05-C-0084 for the Air Force Research Laboratory. This one-year research effort focused on defining the FuseFlow language and architecture, and developing early prototypes of the major components.

3. Methods, Assumptions, and Procedures

3.1 FuseFlow Architecture

FuseFlow is envisioned as an integrated system for fuselet workflows which handles their complete lifecycle. In order to handle the runtime aspects of fuselet workflows, this system must therefore interact with a running JBI and its fuselet server and metadata schema repository (MSR)

Authoring of fuselet workflows requires assignment of a fuselet class to each task[1]. Such assignment requires (roughly) the information that is in the fuselet class metadata. If authoring occurs in conjunction with a "live" JBI, then access to the fuselet repository will provide information about all fuselet classes that are available for use in the workflow. Authoring also requires access to the type hierarchy (as in the MSR).

Authoring "offline" may be a more common occurrence. In this use-case, the collection of fuselet classes and managed information object types can be separate from the JBI, and might be part of a fuselet authoring environment. In either live or offline use, *fuselet authoring* is expected to be an associated activity. Fuselets and the workflows that employ them will most likely be created together. Thus, integration of fuselet *workflow* authoring with a fuselet authoring environment is desirable. The FuseFlow architecture envisions such integration occurring with the iFUSE[2] authoring environment.

One of the chief advantages of workflow management is that it provides detailed visibility into the actual functioning of the process. Logging this information and analyzing it in conjunction with a record of performance outcomes can suggest areas needing improvement and strategies for effecting this. The FuseFlow architecture has provisions for this feedback loop. This report discusses those "process improvement" activities and what integrated support for them might be like.

The FuseFlow architecture is shown in Figure 1. The system is shown layered on top of the JBI, as discussed in the previous paragraphs.

3.1.1 FuseFlow Authoring and Compiling

This component is used to write process definitions in the FuseFlow language, and to prepare them for execution (i.e. compile them). The editor is graphical, enabling the user to sketch the major features of the process as a diagram. From this diagram, dialog boxes can be opened to

---

[1] Actually, only some of the tasks will be carried out by fuselets, requiring such an assignment. Other tasks, carried out by people using JBI clients or implemented as subflows do not have an assigned fuselet.

[2] iFUSE was developed by ATC-NY on an AFRL SBIR effort.

specify the details of the process. In FuseFlow the scope of what is specified via this interface is somewhat broader than in classical workflow management systems: The specification can contain items that will drive process monitoring and will contribute to process improvement.

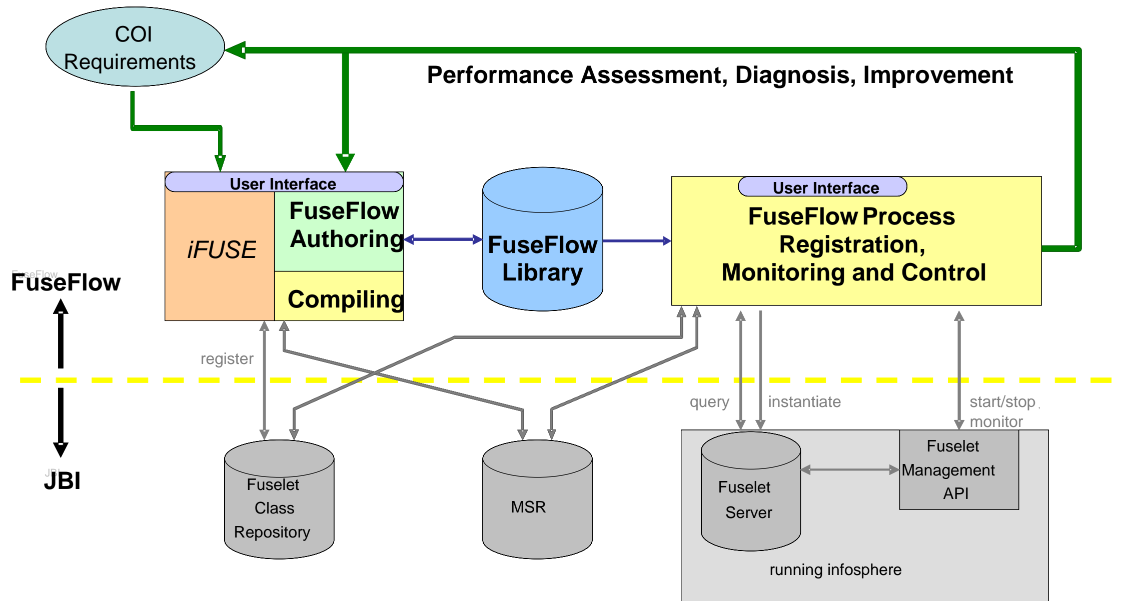

Figure 1. FuseFlow System Architecture

A shared interface with the iFUSE [6] fuselet authoring and analysis environment permits smooth transitions between working on fuselets and on their containing workflows. As mentioned above, the authoring environment also shares with iFUSE its view of what the fuselet classes are, and what the managed IO types are.

Processes will be enacted by collections of fuselets, as described in detail in Section 3.4. Compiling converts an abstract specification of the process into a specification of the fuselets to run and the parameter values to supply them with.

FuseFlow specifications are envisioned as having formal parameters to make them more reusable. Compiling will then instantiate the specification using actual parameters (values).

The artifacts of authoring and compiling are placed in the FuseFlow library.

The discussion of the FuseFlow authoring prototype is deferred until after the explanation of the FuseFlow language; see Section 3.3. The details of the compiler prototype are described in Section 3.4.3 after process enactment via fuselets has been explained.

3.1.2 FuseFlow Process Registration, Monitoring, and Control

As mentioned in the previous section, a compiled process is, in effect, a specification for a set of fuselets to run. The *Monitoring and Control Client (MCC)* registers those fuselets, runs them, monitors their execution, and eventually stops them. A user interface allows the information management staff to control these actions and view the process status.

The Fuselet Management API (FMAPI) is used to interact with the fuselet subsystem of the JBI. This allows the MCC to interact with any compliant JBI or (separate) fuselet server.

The work carried out by such processes is divided into cases, as described in the background section on workflow management, Section 2.1.2. Monitoring and control is applied to the workflow as a whole, to the individual fuselets that comprise it, and to the cases that are executing or are completed. So, for example, the interface shows which cases exist and what their status is. For each case, it shows which tasks have been started, which have completed, and whether an error occurred in processing.

The FuseFlow language allows the specification of workflow outputs that are to be monitored (the "tests"). These are described below in Section 3.2.5. Tests are Boolean values that are computed per case and are intended to reflect significant domain-specific conditions pertaining to the case. For example, a test may report the result of a "sanity check" which should be true of all cases of the workflow. The monitoring and control component should be able to retrieve the test results and display them in various formats (e.g. highlighting cases for which the test yields the value FALSE).

The MCC prototype is an Eclipse plug-in which presents a user interface in the Eclipse Workbench. As suggested by the architecture diagram, it is both a CAPI client and a FMAPI client. The FuseFlow library is implemented as an extension of the FREeME [7] fuselet library, and is accessed via a web services interface. A screen shot is presented in Figure 2.

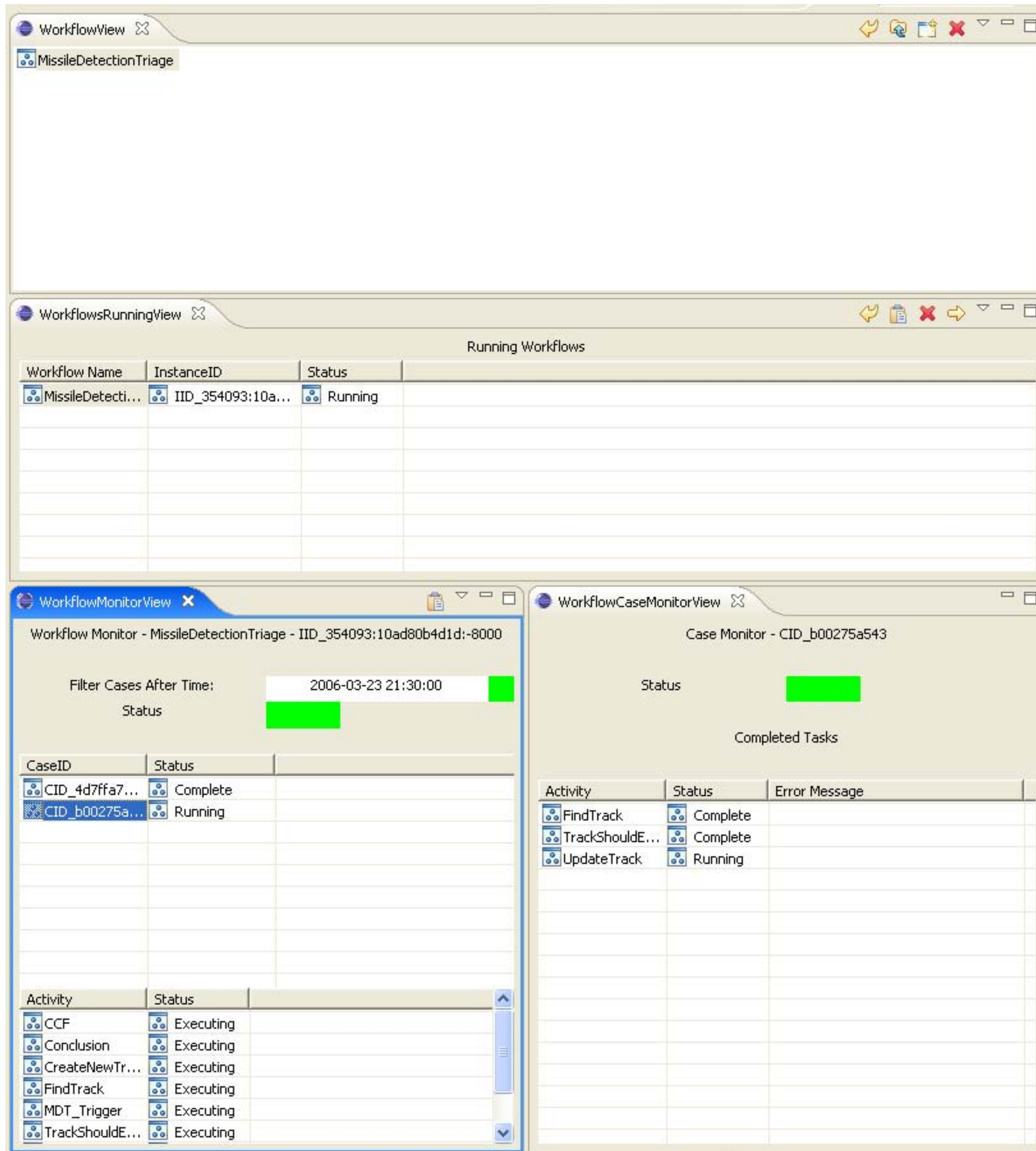

Figure 2. Monitoring and Control Client user interface

The upper pane presents a view of the FuseFlow library. In this example the library contains one workflow (compiled). Buttons across the top will refresh the view and load and start the selected workflow. This includes loading all of the required fuselets into the fuselet server. The fuselets' instance metadata is constructed on-the-fly and the parameter values are thereby supplied. The required fuselet classes must already exist in the fuselet repository.

The next pane down shows the instantiated workflows, in this case just one. Buttons across the top of this pane can stop the workflows and can retrieve previously run workflow instances.

The bottom pane on the left shows one of the workflows being monitored. At the very bottom are the fuselets involved, together with their statuses. Just above this are the cases that this workflow has executed and the overall workflow status. There are two cases shown; one completed and one still running. The overall workflow status is indicated by the green block, meaning "running."

The status of one of the cases is displayed to the right. It shows the tasks that have been started for this particular case, and their current status. The overall status of this case, indicated by the green block, is "running."

The MCC prototype is at an early stage of development. It illustrates the concept of a display that supports "drill down" – in this case, from the workflow to the instance, then to the cases, and finally to the tasks of a case. However, one could certainly drill down further: the information about how each task ran is present, but not displayed. Furthermore, the objects that each task consumed and published are available, and could easily be retrieved.

Drill down will be essential for information management staff to understand what their processes are doing, and especially for diagnosis of problems. One of the barriers to an effective drill-down capability is the difficulty in identifying specific cases of interest. If there were many workflows, and many workflow instances, then these might pose a similar problem. Time-stamps and time intervals on processes, cases, and tasks might be a great benefit in this area, and would have other uses as well. If this information could be displayed on a timeline, that would be a particularly suitable interface for accessing this information.

Currently, workflow instances and cases are identified by meaningless unique identifiers. For human consumption, it would be better to fashion "names" using data that is part of the case or instance. For a workflow instance, a name can be solicited from the IM staff as they create it. For cases, the workflow author can select specific items of data that are present in the triggering objects to concatenate into a name.

In the Section on Process Assessment and Improvement (Section 3.5) another approach, using OLAP (On Line Analytical Processing), to finding relevant cases is discussed. OLAP is a data mining technique.

Broader views of the functioning of the workflow are clearly possible and seem desirable, though none of these are implemented in the prototype. As an example, statistics about the rates of occurrence of errors, and graphs or histograms of this error data might make error trends visible – trends which may be diagnostic of problems.

Tests (described above) are not implemented. However, error conditions are displayed on a per-case and per-task basis.

3.2 FuseFlow Language

FuseFlow is a “dialect” of XML whose syntax is defined by an XML Schema. The purpose of this section is to explain the major design decisions leading to this language.

3.2.1 Process Definition

At its core, FuseFlow is a *workflow* or *process definition* language. For this aspect of the language, XPDL [3], an XML process definition language standardized by the Workflow Management Coalition is extensively utilized. An XPDL specification, in general, defines a set of processes or workflows within a larger context (a “package”). The *ProcessDefinition* XML tag and its contents borrow extensively from the *WorkflowProcess* tag of XPDL.

The FuseFlow system is, in effect, a workflow management system using the FuseFlow language for its runtime specifications. Fuselets and other JBI clients carry out the activities in this system.

In FuseFlow, most activities correspond statically to JBI fuselets, referred to here as *task-bearing fuselets*. Each of these fuselets stands ready to carry out one of the activities of the workflow. As each case reaches the stage where a particular activity is enabled, the system signals the corresponding fuselet to run to handle that case.

3.2.2 Control-Flow Logic

The process definition specifies the sequence in which activities should be carried out for each case. Some activities may execute in parallel, and some may only execute if certain conditions are met. The basic structure of a process definition is a labeled directed graph. The nodes of the graph correspond to the activities, and the directed arcs are called transitions. In this system, these graphs are restricted to be acyclic. There are two special nodes, START and END. A single transition leads outward from START, with no inward transitions, and it is the only such node. Similarly, a single transition leads towards END, with no outgoing transition, and it is the only such node. These properties, together with acyclicity, guarantee that the intuitive understanding of “start” and “end” are matched by those nodes – any abstract “flow” beginning at START and flowing along the directed arcs will necessarily terminate at END. Such process definition graphs are often depicted “graphically”. Figure 3 presents a small example:

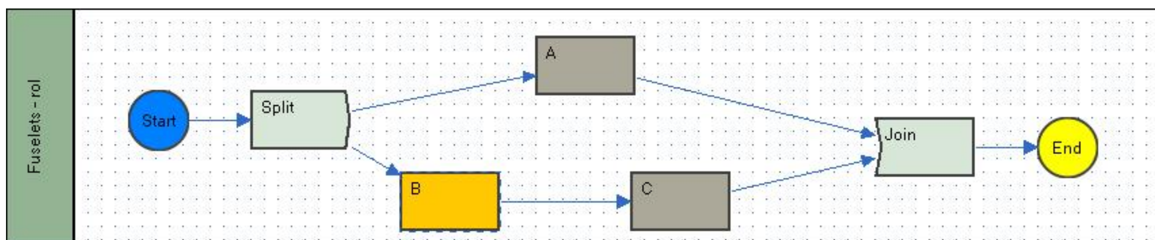

Figure 3. Example workflow diagram

Each task in the workflow may be either a "split," a "join," or both. As shown in the figure below, a split is a task with multiple outgoing transitions, while a join is a task with multiple incoming transitions.

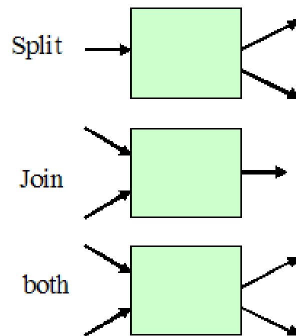

Figure 4. Activities with multiple transitions in or out: splits and joins.

The unique *Start* node and *End* nodes are represented in these diagrams as small circles.

Each split or join is labeled as either **AND** or **XOR**. When a task is both, then each side must be separately labeled. The graphical notation used to distinguish these two labels on the inputs (or outputs) is to use a straight or bent left-hand edge of the box (respectively, right-hand edge of the box). Figure 5 presents the four possibilities:

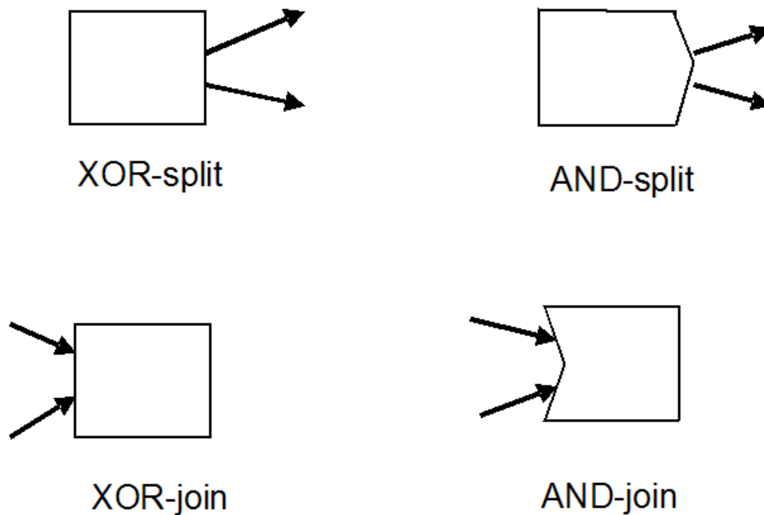

Figure 5. Graphical notation to distinguish AND and XOR control flow.

When there is just one incoming transition, or one outgoing transition, then the distinction between **AND** and **XOR** is moot, and a straight left- or right-hand edge can be used.

The graph structure of the process, the AND/XOR labeling of each split and join, and possible *conditions* on each transition govern the flow of control within each case. A condition is a Boolean expression that may be evaluated with values supplied by completed workflow activities from the same case, as well as external values. Here is how this is done:

When a task completes, one or more of the outgoing transitions (arrows) is *activated*. When incoming transitions of a task are activated, the task may be *ready to start* (see below).

Some or all of the tasks that are ready to start are actually started, and run to completion. Eventually, all tasks that become ready to start should start and complete. However, completion is a property of the task performer (i.e. fuselet) and thus cannot be guaranteed by the FuseFlow system.

Here are the rules for activation at splits, and for task starting at joins:

- AND-Split
  - All outgoing transitions are activated
  - Outgoing transitions may not have *conditions*.
- XOR-Split
  - Exactly one outgoing transition is activated
  - Activate the first transition whose condition is true (or that lacks a condition).
  - If no transitions lack a condition, then there must be one transition with the special condition *OTHERWISE* that is activated when all others are ruled out.
- AND-Join
  - All incoming transitions must be activated to start the task.
- XOR-Join
  - Any incoming transition (activation) will start the task.
  - All additional incoming transition activations are errors.

There are two kinds of tasks or activities that need to be considered: generic tasks and *routes*.[3] *Routes* are dummy tasks whose only processing entails transferring control from incoming

[3] XPDL defines several other kinds of tasks (e.g. block) and several variants (e.g. loop) that future revisions of FuseFlow may adopt.

transitions to outgoing ones. They are used to implement more complex control flows using the control flow primitives described above. Routes are an abstraction which may be implemented differently than other tasks for efficiency. Figure 6 presents an example of a Route used in a fragment of a workflow:

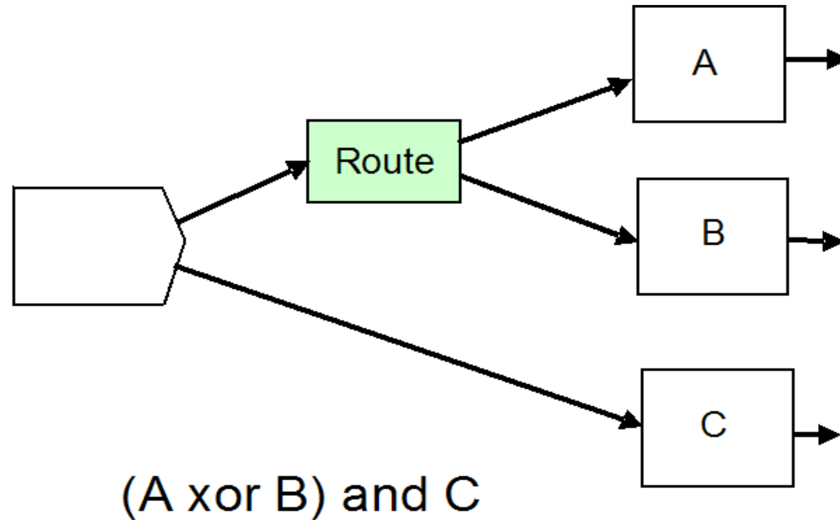

Figure 6. Using a Route in a workflow to express complex control flow.

In this example, after the left-most activity completes, C will be ready to run, and exactly one of A and B will be ready to run. The choice between A and B would be based on conditions on the transitions from the Route to A and to B, if any. Since the left-most activity is an AND-split, its outgoing transitions cannot have conditions.

3.2.2.1 Control Flow: The road not taken

XPDL has a more general control-flow model than FuseFlow. The FuseFlow subset was selected as being both parsimonious and precise. In XPDL, the author declares that the specification as a whole will be *full-blocked*, *non-blocked*, or *loop-blocked*. These choices express constraints on the workflow diagram. Full-blocked means that the diagram must be strictly nested, with each split having a corresponding join of the same type. Loop-blocked means that the diagram must be acyclic. Non-blocked means that the diagram can have an arbitrary shape.

In XPDL, for non-blocked and loop-blocked specifications, the transitions leaving AND-splits can have conditions. Their meaning is, roughly, that all transitions leaving an AND-split with a condition that evaluates to true (or with no condition) are activated. The specification of what happens at AND-joins in these circumstances is ambiguous and very confusing.

The following conclusions were developed:

1. Conditions on AND-split transitions are unambiguous only in the full-blocked mode.
2. Conditions on AND-split transitions are not needed for full generality in any mode.

Thus, the only situation when XPDL allows AND-split conditional transitions is when they are not well defined! Even when they are well defined, they aren't needed..

In light of these considerations, the loop-blocked restriction was adopted, but only conditions on transitions that leave XOR-splits are allowed.

3.2.3 Declarative Structure

The externally visible behavior of a fuselet workflow consists of triggering conditions (i.e. information objects), other input and output objects, and error conditions that may be signaled. This *necessarily visible* behavior is distinct from the manner of implementation (e.g., via specific fuselets) and its visible manifestations. For example, objects published as part of workflow coordination would fall into the latter category.

FuseFlow has several features designed to separate out such "declarations" from the "implementation" part of the specification. The items just mentioned are all specified in separate XML elements that are directly contained in the top-level FuseFlow element. A sibling element, ProcessDefinition, describes the activities and transitions – in effect, the implementation. All of these elements are "optional" in the schema. What's more, these elements may appear in any order.[4] The result is a highly flexible and forgiving syntax. This supports a design-implement-deploy-redesign lifecycle: At the design stage, a specification for what the workflow is supposed to accomplish is still syntactically legal. The ProcessDefinition element can then be added during implementation. During redesign, an improved implementation might be specified without re-declaring the external behavior – again a legal FuseFlow document.

This syntactic distinction between declaration and implementation will promote reuse of specifications (i.e. by those who do not care much about how they are implemented, just what they accomplish). It also enables analysis of information flows within the infosphere at a larger granularity than individual fuselets and clients.

All of this syntactic flexibility does not absolve us of the need to have complete specifications whose parts correctly refer to one another (i.e. for compilation). However, these properties are relegated to static semantic checks that are performed by the relevant tool at the appropriate moment. This approach has the advantage that the checks may be context dependent, are only performed when needed, and may result in more informative error messages.

[4] The complex type of the FuseFlow element uses the "all" model.

3.2.4 Data Fields

Task-bearing fuselets, in performing their activities, will receive information objects via subscription and query, and will publish other objects back to the infosphere. These *functional objects*[5] carry information between fuselets and also between the workflow and clients outside the workflow.

The workflow automation system at the heart of the FuseFlow runtime is concerned with managing the flow of control between fuselets, and not the flow of data. Thus, it treats the functional objects as opaque – it is aware of the types of objects that should be published within each activity, and the ObjectID of the actual objects in each case. However, it never tries to obtain those objects, nor does it have code that would enable parsing of their metadata or payload.

Yet, the workflow automation system needs some portion of the information created during task execution for control and for tests (see Section 2.4). When there is an activity with an XOR-split, the decision on which of the outgoing transitions to activate will depend on evaluating Boolean expressions called conditions that are associated with each transition (see Section 3.2.2). This is mostly of interest when the decision for each case can be affected by what has already happened in handling that case. Thus, the conditions must depend on the results of task execution, i.e. objects that have been published. The solution adopted here (and in XPDL) is that conditions depend on data fields – essentially typed variables that are declared as part of the workflow.

So, how do data fields get their values? A side effect of executing certain tasks is that these values are extracted from the results and "exported" to the workflow management system. The FuseFlow language enables specification of which tasks can export values for which data fields, but not how to do the extracting.

An important point about data fields in FuseFlow (as distinct from XPDL) is that they can only get a value once per case. This avoids certain very hard problems in concurrency. It also makes data fields more "declarative" and should be easier to understand and audit.

For a condition to be evaluated, the data fields mentioned in it must have assigned values. A workflow is incorrect if it is possible to complete a task with an XOR-split, and then not be able to evaluate the conditions on outgoing transitions.

Both the top-level FuseFlow element and the ProcessDefinition element have a place to declare data fields. The idea is that some of these values are important parts of the "result" of the workflow and would be declared in the outer scope. Other data fields are only meaningful when considering how the workflow is implemented, and are declared in the inner scope. The first

---

[5] Functional objects are distinguished from *control objects* that are used to coordinate the flow of control between fuselets.

group of data fields could be used for tests or conditions, while the second group will only be used for control-flow conditions.

3.2.5 Tests

Tests are Boolean expressions that depend on data fields. They are similar to conditions, but they are declared in the outermost scope of FuseFlow, and they can only use the data fields that are declared there.

The runtime system (the workflow automation system) monitors the data fields that occur within each test. When all of the necessary data fields have assigned values (for a particular case), the expression is evaluated, and the result is published into the infosphere. Because data fields can only be assigned values once, the result of the test cannot change during subsequent execution of the workflow (for that case).

Tests are intended to identify the cases of the workflow that fall into externally meaningful categories (e.g. success or failure). The FuseFlow monitoring and control client will make use of this information in order to support monitoring of the running workflow. For instance, it will be able to display the number of cases in each category, and identify specific cases in a category. The design of these categories, and of the corresponding tests, will typically require subject-matter expertise. Thus, it is appropriate to let the workflow designer (who has that knowledge) specify them. The monitoring and control client on the other hand will be generic – capable of displaying any test results.[6]

3.2.6 Error Handling

Errors may occur during task execution and within the mechanisms for control flow. Some errors are quite generic (e.g. some task never completes) while others are specific to the particular workflow. FuseFlow has provisions for declaring the error conditions that can occur in a workflow, especially this latter kind. It also allows for the specification of which activities in the workflow (if any) can raise those exceptions.

The fuselets which participate in the workflow will report their status to one another and to the FuseFlow infrastructure by publishing objects. When errors occur, those conditions are marked in the corresponding status object. Status objects are described in more detail in Section 3.4.

Error *handling* is processing that happens in response to an error in order to mitigate the problem. The approach taken is that error handling will be specified in a separate workflow that is triggered by the published error-containing status objects. Such workflows might roll back incomplete transactions, restart failed computations, notify human operators, or create log entries.

---

[6] The client will also be able to display the values of data fields for each case, or derived statistics.

3.2.7 External Context

FuseFlow specifications can record the *intended* external context of a workflow. This context includes the producers of objects that are needed by the workflow, the consumers of objects that the workflow publishes, and other entities that are concerned about the workflow – the stakeholders.

The purpose of these specifications is to enable better management of the workflow, and to contribute to process improvement. For example, by knowing who the intended consumers are, managers can better understand the consequences of shutting down or modifying a workflow. Managers can also compare the intended consumers with the actual consumers, to see if underlying design-time assumptions are still valid. Stakeholders (i.e. people) may be queried to see if they are satisfied with the output of the workflow, in an effort to improve it.

3.2.8 Performance Dimensions

One focus of the FuseFlow project is evaluation of the performance of fuselet workflows, diagnosis of problems, and redesign to address those problems. As with conventional workflows, there are some standard measures of performance such as throughput and latency. Analogous to cost, similar measures of the resources can be used to handle cases. However, there is still the important matter of the quality of output and its relevance to the end user. Of necessity, these aspects will be specific to the subject matter and will be multi-dimensional. The approach is to enable specification of *performance dimensions* for the workflow, and appropriate means for grading each case along each dimension.

Performance dimensions are analogous to categories of defects or flaws in manufactured output. A manufacturer will certainly devise a taxonomy of defects and will attempt to discern the rate of occurrence each for category by various means. The kinds of defects that can occur will depend on the specific product (e.g. blemished paint job). Similarly, defective FuseFlow cases (in effect, defective information objects) will fall into meaningful domain-specific categories. This puts the burden on the workflow designer who should be a subject matter expert.

The means for grading each case with respect to particular dimensions will fall into one of several general categories, reflecting different sources of evidence for performance. Some of the tests (see Section 3.2.5) or errors will provide internal evidence for success or failure. Participants (humans carrying out workflow steps) or stakeholders may also be asked about the case to obtain subjective evidence. An expert may also be asked to examine specific cases, or a sample of cases to obtain expert judgment. There may be an external result, reflected elsewhere in the infosphere that provides evidence of success. Finally, there may be some other "downstream" workflow whose success is dependent on high quality outputs from the subject workflow. This aspect of the specification is still not fully worked out – the specification contains place-holders for each of these categories.

3.2.9 Formal Parameters

FuseFlow specifications will have formal parameters which must be given values in order to create a corresponding fuselet workflow. This will allow FuseFlow specs to be reused to start multiple workflows. This aspect of the FuseFlow language will be fully developed in a future version.

The idea is that the formal parameters are used to condition expansion and substitution within the specification. This will require some extra syntax (elements) for variable substitution, conditionals and loops within this expansion – essentially a macro language. The finished (expanded) specification does not contain any of these constructs, if successful – it thus has a more straight-forward interpretation.

3.2.10 Activity Specifications

The specification of each activity with the ProcessDefinition element will contain the control flow and external behavior of the activity, such as objects published and errors that may be raised, as discussed in prior sections. Part of the Activity element will also specify how it should be implemented (e.g. by a fuselet, or by a conventional client). For activities that are to be performed by fuselets, the fuselet class can then be specified, along with any necessary instance parameters.

Thus, the external behavior specification of an activity corresponds closely to the fuselet class metadata, and might be the basis for searching for an appropriate fuselet class to perform an activity. The implementation specification corresponds to fuselet instance metadata, and thus enables instantiation.

3.3 FuseFlow Authoring Tool

The purpose of the FuseFlow authoring tool is to enable easy generation of syntactically correct FuseFlow process specifications that expresses the intentions of the author. It is also capable of static semantic checking to flag some specifications with certain subtle problems.

The FuseFlow authoring tool prototype, called the *FuseFlow Process Editor (FPE)* is an Eclipse graphical editor. It runs in the Eclipse workbench, and can coexist with the iFUSE fuselet authoring environment. It shares some code with iFUSE and, as a result, it uses the same local Metadata Schema Repository. So the FPE and co-resident iFUSE will share the same view of what the object types are. Although it was not explored, it should be straight forward to implement sharing of iFUSE's view of fuselet classes as well.

In Section 3.2.2 a graphical notation for workflow processes is described. The FPE uses this same notation as its key view. Figure 7 shows the FPE open in the Eclipse workbench. The central window is the graphical editor for workflow diagrams. In the figure, the numbers on (or near) transition arrows indicate the priority order in which they will be considered for a transition from an XOR split. If the split is AND, or if there is only one outgoing transition, then the

numbering is irrelevant. On the right-hand side of this editor window is the Palette that is used to add new activities and transitions to the diagram. Selected activities and transitions can be deleted from the Edit menu, or from the right-click context menu. The graphical editor supports a full undo/redo stack.

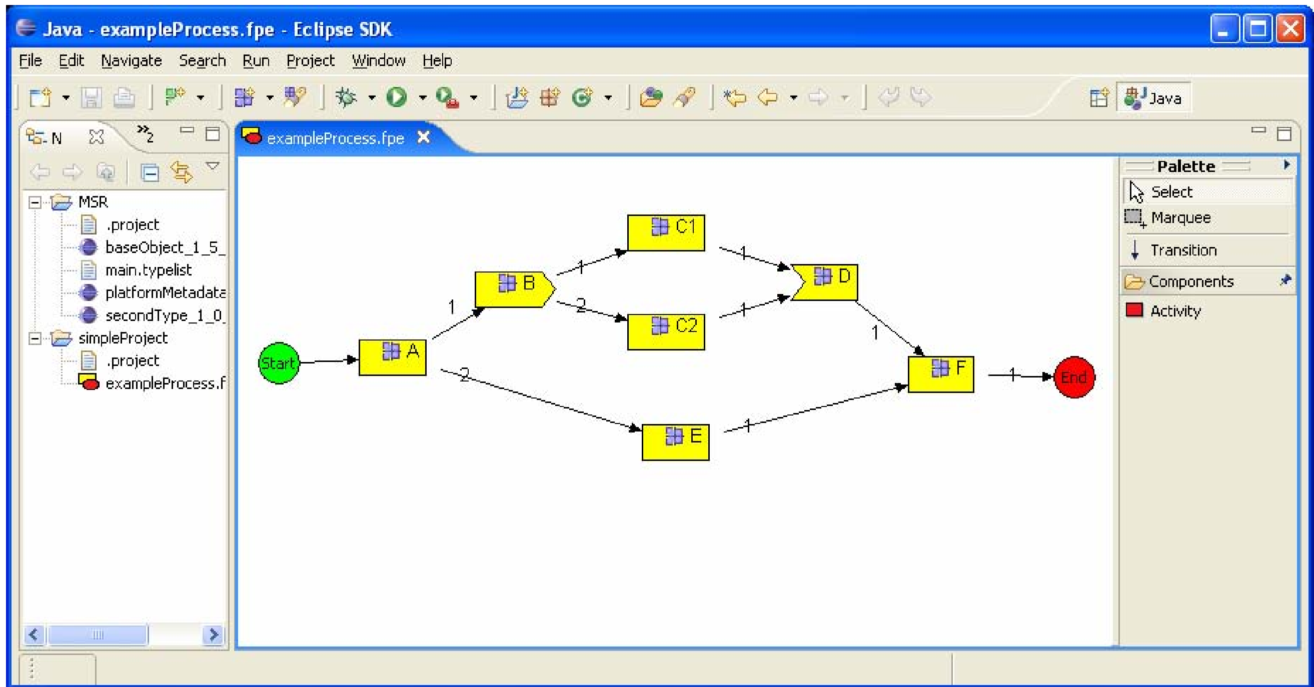

Figure 7. FuseFlow Process Editor

The icons inside of each of the yellow Activity symbols indicate that each of these is to be implemented via a fuselet. The editor supports several other implementation choices, though these do not (yet) have an icon to place inside the symbol.

On the left-hand side of the figure, the Eclipse Resource Navigator is open. It shows two projects. The one named "MSR" is the iFUSE Metadata Schema Repository. The other project contains the workflow process that is being edited.

This editor uses dialog boxes to configure other aspects of the process. Double-clicking on an activity brings up the dialog shown in Figure 8. This shows the properties of the "A" activity from Figure 7. Radio buttons are used to select the type of activity (i.e. how it is implemented). Radio buttons below that are used to select the split type and join type. There is also a pane to view and adjust the order of the outgoing transitions which are referred to by their destinations ("B" and "E" in this example).

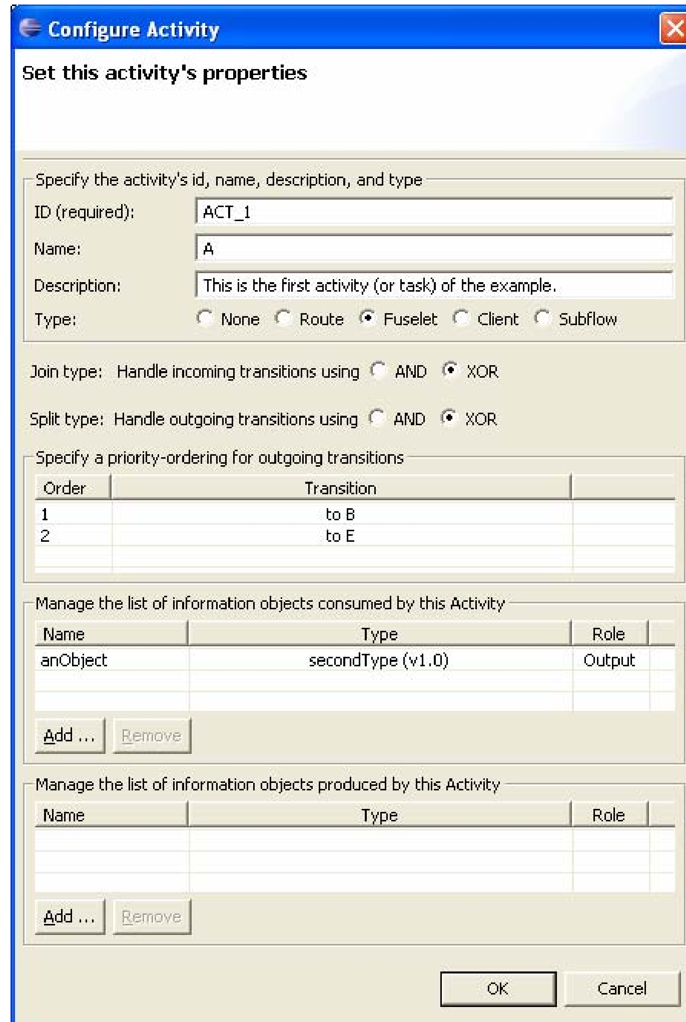

Figure 8. Setting the properties of an activity

The FuseFlow language has provisions for specifying the signature of activities – the objects that are input and those that are published. The next two sections of the dialog box are for viewing and setting these inputs and outputs. Note that an input (object) of an activity may have been published by some other activity of the workflow. Such an object is therefor an output of the workflow as a whole. This distinction is the meaning of the "role" column in these list boxes. Each object also has a name (shown in the left-most column). This name is, in effect, a formal parameter of the generic workflow case, referring to a *potential* input or output object of the case. This is explained further in Section 3.4, below.

The "Add" button gives access to a dialog box showing all of the objects that may participate in the workflow, together with their names, types, and roles. An object may be selected from this list, or a completely new entry may be created – using an object creation dialog that then appears. To create a new object (actually, a new formal parameter that will refer to an object),

the user must specify its type. The known types from the Metadata Schema Repository (MSR) are displayed for selection. The user may even create a whole new type, which is then added to the MSR. These dialogs are straightforward, and so are omitted here for brevity. The net result is that the signatures of the activities and of the whole workflow may be specified, and that these will be consistent with one another and with the MSR.

Double-clicking on a transition arrow brings up a dialog box for specifying the condition on the transition. A condition is a Boolean expression which governs whether this transition will be taken or not.[7] In the prototype system, legal conditions are any Jython expression or the special string OTHERWISE. If the expression evaluates to True or non-zero, then the transition may be taken. The variables that may appear in condition expressions are explained in Section 3.4 below.

The editor context menu gives access to the global list of objects mentioned above, as well as a dialog box that "configures" the workflow. The name of the workflow is set here. More significantly, this dialog has a button that causes the process specification to be written out in FuseFlow (XML) form. An edit box enables setting the name of the output file. The "native" file format which the editor creates and saves is an internal format, different from the FuseFlow language.

The FPE can be used to author specifications with a significant fraction of the FuseFlow language features, but not all of them.[8] For example, performance dimensions and tests cannot be specified with the editor. In order to use the editor in conjunction with the other prototype tools, there are two aspects of the FuseFlow specification which must be edited by hand, since these are not implemented in the FPE:

- The name of the fuselet class which will implement each activity (the Task-Bearing Fuselet, described in the next section) must be specified, and[9]

- The class name and parameters of the Trigger fuselet.

With these adjustments, the FuseFlow specification can be compiled, loaded, run, and monitored with the prototype tools.

---

[7] Conditions are only useful for transitions out of an XOR split, with multiple transitions.

[8] The output is syntactically legal because the omitted features are all optional.

[9] Parameters of the fuselet that do not pertain to the FuseFlow infrastructure may also be specified.

3.4 Process Enactment

Fuselet workflows are enacted by interacting fuselets. The fuselets publish and subscribe-to a suite of *control objects* which automates the flow of control among the fuselets. The control objects also carry information about task results, including the IDs of published "functional" objects. This enables the flow of data between fuselets via queries.

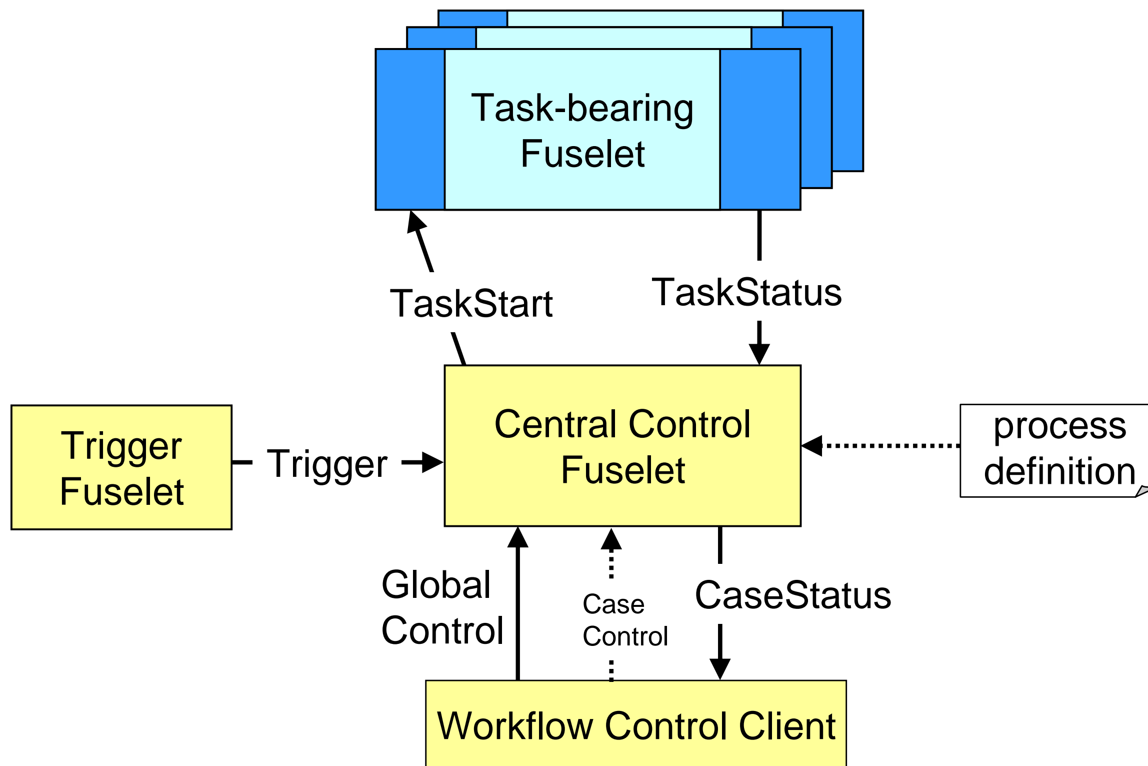

Figure 9. FuseFlow Runtime Architecture

Figure 9 illustrates the FuseFlow runtime architecture – the way fuselets and information objects are employed to enact the workflow. In this figure, the boxes represent fuselets (and one client). The labeled arrows represent control objects that are published by one fuselet or client and subscribed by another.

3.4.1 Tasks and the Central Control Fuselet

There is one fuselet to execute each task (or activity) of the workflow. These are referred to as *Task-Bearing Fuselets (TBFs).* Another fuselet, the Trigger fuselet, detects the conditions for a new case of the workflow, and signals the others. The *Central Control Fuselet (CCF)* keeps track of the cases of the workflow and which tasks have already been completed for each case. Following the process definition, the CCF signals each TBF to run when its preconditions are met. All of this signaling and fuselet processing is per-case.

All of the signaling between these fuselets occurs via the publication of (and subscription to) control objects. Control objects are ordinary JBI objects. The principal types of control objects in the prototype are: *Trigger*, *TaskStart*, *TaskStatus*, *GlobalControl*, and *CaseStatus*. The Trigger fuselet publishes a Trigger object to signal the creation of a new case. The Trigger object contains the *Unique Identifiers (UIDs)* of the objects that caused the trigger. The CCF subscribes to the Trigger objects and creates a new case whenever one is received.

The CCF publishes TaskStart objects to signal TBFs to run. The TBFs, in turn, publish TaskStatus objects to signal the CCF that they have finished executing or have experienced a fatal error. The CCF also publishes a CaseStatus object, summarizing the status of the relevant case, each time a TaskStatus object is received.

Different cases of the same workflow may coexist within the "system" of fuselets, and may execute concurrently. This is possible because each case has an assigned UID, which is embedded in all control objects (except the Trigger).

Multiple fuselet workflow instances can also coexist without confusion. Each workflow and each instance are assigned a UID which are embedded in all control objects (including the Trigger). The fuselets are all handed the workflow UIDs as an initialization parameter. The UID is used in the predicates for all subscribed control objects. Thus, the control "signals" between concurrently executing workflow instances are strongly separated.

The other important aspect of fuselet workflows is "on demand" data flow. TBFs can publish objects containing the results of their processing (the transformed information). Downstream TBFs *working on the same case* need to input those objects for further transformation. The mechanism for enabling this is illustrated in Figure 10. The process definition assigns an object name (i.e. a formal parameter) to each possible object that *may* be published in handling a generic case of the workflow. Whenever a TBF publishes an object, it records the UID of the object and its name in the TaskStatus object that is then published. The CCF builds up a mapping between names and object UIDs (separately for each case), and sends this mapping to each downstream TBF as part of the TaskStart object. The downstream TBF, requiring one of the named objects, can easily obtain it by a query on the corresponding object UID. The effect of this mechanism is convenient on-demand flow of information objects, strongly separated by workflow case.

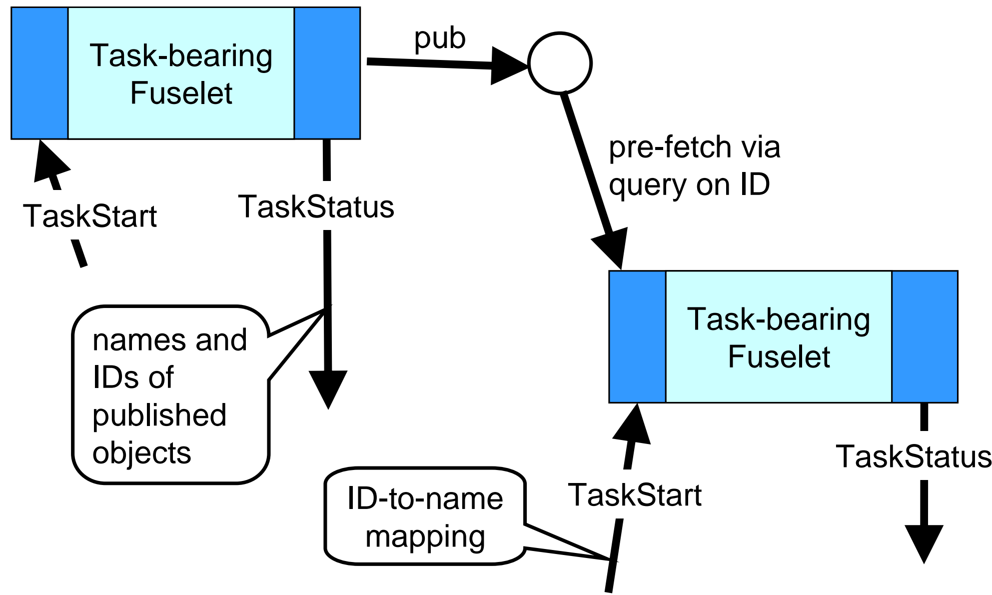

Figure 10. Data flow between workflow fuselets

When the process definition calls for an XOR split, the CCF may need to evaluate *conditions* on the outgoing transitions in order to select one for activation. A condition is a Boolean expression. The prototype CCF supports any legal Jython expression, treating any non-zero or non-null result as *True*. Expressions are evaluated in the context of variable bindings that are supplied by the TBFs. The FuseFlow process definition may contain declarations of variables that can be bound by TBFs[10] called data fields (see Section 3.2.4). Variable bindings supplied by the TBFs are carried to the CCF in the TaskStatus objects that they publish. The CCF maintains a separate context for each case, and enforces that each variable can be assigned at most once during each case.

Tests are similar to conditions: they are expressions that are evaluated in the context of per-case variable bindings. They are evaluated in the CCF. However, unlike conditions, their values are published in the infosphere. The Monitoring and Control Client can then use those values, which are associated with specific cases, to facilitate selecting among cases.

The prototype implementation of this enactment method consists of the CCF and a template for TBFs, together with schemas for all of the control objects. Both the CCF and the TBF template are implemented in Jython, and run correctly with the FREeME fuselet execution environment. The TBF template may be extended by subclassing to add code to perform the meaningful part of

---

[10] In the current prototype, these declarations do not actually constrain the names or types of bindings, though that is the intention.

each task. The template takes care of many details such as subscribing to TaskStart, publishing TaskStatus, pre-fetching named objects, and correct multi-threaded access to query sequences.

The prototype does lack several intended features, though all of these are straightforward to implement. These include optimized processing for routes, evaluation of tests, and evaluation of transition conditions in the correct order. There are a number of fairly obvious additional features that could also be contemplated, such as using the Activity signature to proactively create query and publisher sequences.

3.4.2 Alternative Enactment Architectures

A number of alternatives were considered when designing the process enactment architecture. Each of these had some advantages, and some disadvantages.

In this first variant, activities subscribe directly to the TaskStatus objects of their direct predecessors in the workflow. Each activity must then implement the join semantics (either AND or XOR) on its incoming transitions. Activities must also implement the evaluation of conditions. This could happen at either end of the transition. The primary advantage of this scheme is that there are fewer objects being published; roughly, collapsing TaskStart and TaskStatus into one type. In circumstances where fuselets are publishing "functional" objects that can be easily identified (with respect to case), the TaskStart might be dispensed with altogether. This holds out the potential to integrate some ordinary fuselets that are not aware of FuseFlow into a workflow.

There are two primary disadvantages of this variant compared with the CCF-based method. First, most fuselets must have knowledge of their predecessors or ancestors in the workflow (or both). They must also know about their own split and join semantics, including possibly conditions on transitions. Naïvely implemented, such fuselets will be far less general than those which participate in the CCF scheme. Of course, a TBF template can be developed that can handle all of this (in general), but that is a lot of machinery to put in each task-bearing fuselet, and would require much more complex parameters for each fuselet.

The second disadvantage of the variant architecture arises due to the fact that FuseFlow control semantics require persistent state. In the CCF scheme, this state can all be in the CCF, while the TBFs can remain stateless. For the direct-interaction variant, must be present, to some degree, in all of the TBFs, with some portions of the state replicated in multiple fuselets. To accomplish this, the state (or portion) must be published into the infosphere, and subscribed. Getting all of this to work seems much more complicated than putting the whole state and all of the code that acts on it in one fuselet.

A third scheme employing multiple, smaller control fuselets, roughly one per transition was also considered. This was quickly rejected, since it has all of the disadvantages and few of the advantages of the first variant. It requires a larger number control objects to be published, as with the CCF, but also requires a much larger number of fuselets. It takes some of the contextual

knowledge out of the TBFs, but not all of it! It does remove the state from the TBFs, but fragments it in many control fuselets, resulting in inefficient controls and error handling.

3.4.3 FuseFlow Compiler Prototype

The FuseFlow compiler prepares a FuseFlow specification for loading and execution. It maps the specification into a complete set of fuselet instances to enact the workflow, as described in the previous section.

The compiler prototype is a python program that is run from the command line with two parameters: the FuseFlow input file and an output file name. The output file, which should have a ".WF" extension, contains a list of all of the fuselets in the workflow, and, for each fuselet, a complete list of all parameters. A few of the fuselets, with special roles (first, last, trigger, CCF) are specially marked. The Monitoring and Control Client follows the prescription in the WF file when loading and starting workflows.

The parameters that the compiler specifies for each fuselet are a combination of those directly specified in the FuseFlow input, and some that pertain to the infrastructure. In this latter category are: DesignID, InstanceID, FuseletID, WorkflowName, and LogLevel.

The DesignID and WorkflowName are the identifier and name that are in the Header element of the FuseFlow specification. With the Process Editor these two values can be set in the "Configure Process" dialog available from the context menu.

A FuseFlow specification may be instantiated multiple times, with instances possibly running concurrently. In order to keep them separated, the compiler generates and assigns a UID for the instance, and supplies this InstanceID parameter to all of the fuselets. This parameter value is included in all control objects, enabling fuselets to ensure that they are only responding to control messages from other components of the same fuselet workflow instance. Each time the compiler is run, a fresh value is generated.

The compiler also generates a UID for each task-bearing fuselet. Each TBF is given its own FuseletID parameter. Since this UID is included in the TaskStart and TaskStatus objects, fuselets can subscribe to precisely those TaskStart objects that are relevant.[11]

One of the parameters to the CCF is the process definition in a format that is close to XPDL. (This parameter value may be quite a large string.) For each activity, the corresponding FuseletID is embedded in the process definition. This allows the CCF to publish appropriately labeled TaskStart objects, and to associate TaskStatus objects with the correct activity.

---

[11] The Monitoring and Control Client *also* generates these UIDs fresh, each time a workflow is loaded. Clearly, this is redundant. In a more integrated system, only one of these generation steps would survive.

The vision for the FuseFlow system would integrate the compiler (as with any modern interactive development environment) with the Process Editor. Compiling would happen either on demand, or in response to a build command. The relationship between the FuseFlow specification file and the compiled version would be maintained for future use.

As described in Section 3.2.9, FuseFlow specifications could contain formal parameters that would need to be supplied for instantiation. This would make the facility for multiple instances of the same FuseFlow specification much more attractive. Note that this part of the FuseFlow language is not yet completely specified. It is the compiler (or a pre-processor) that would substitute the actual parameter values and carry out any expansions that are called for.

3.5 Process Assessment and Improvement

One of the primary advantages of conventional workflow automation is that the transparent and persistent record generated by a WfMS can enable a principled approach to performance assessment and improvement. It is reasonable to try to capture these same benefits for fuselet workflows. This effort studied how this might be done.

A model of fuselet workflow performance was adopted that is somewhat analogous to models for manufacturing performance. Provisions in the FuseFlow language are intended to express these models, and thus to support the improvement cycle. The form of integrated support and tools thatwould be required or desirable in this activity was considered. Finally, a list of generic ways in which information spaces can be deficient, and, to the extent possible, a prescription for how to diagnose and remediate them were developed.

In a manufacturing process there are three kinds of "signals" that will typically be considered in assessment and improvement: failures, defects, and perceived value. Failures are events that occur during the process that are recognized as bad. Defects are some aspect of the resulting product that affects its usability. Perceived value is the opinion of the consumers of the product (or the intended consumers) concerning the worth of the product and its suitability for them.

In FuseFlow, the "product" is the set of objects that are published into the information space as a consequence of a case. Failures (in this context also known as *errors*) are exceptional conditions that arise in executing the workflow, either in the FuseFlow infrastructure, or in the task-specific code. Defects, on the other hand, are "something wrong with" those produced objects. Downstream of the workflow there are processes (with human stakeholders), client programs (operated by people), and externally visible outcomes which may be manifestly "good" or "bad."

The FuseFlow infrastructure provides for error signaling by embedding error codes in the TaskStatus objects. These are used to display an error status in the MCC and within the CCF to prevent the start of new tasks for the affected case. Assuming that the TaskStatus objects are persistent, this record of errors will be available for offline analysis.

Section 3.2.8 describes *performance dimensions* which are declarations within a FuseFlow specification of particular ways in which the cases of the workflow can be considered to be defective, or not. Certainly one can expect these to be domain-specific. Thus, it is appropriate to harness the expertise of the workflow author (presumably a domain expert) to specify them. For each performance dimension, the "measures" that can be used to estimate whether a given case is good or bad in that sense can be specified. These measures can include internal consistency tests, the opinions of downstream stakeholders, and downstream positive or negative outcomes. This form of specification prepares us to classify the workflow cases (or a sample of cases) as defective or not. Defect rates and trends can then be considered. It may also be possible to trace defects back to their root causes.

Errors in processing may result in defects, or may simply result in wasted effort and the requirement for rework. The persistent control objects generated by the runtime architecture constitute a detailed record of the work performed, time taken, and (indirectly) the resources consumed in each case. Therefore, the costs associated with process errors (failures), and with normal processing, can be accounted for.

Having understood the workflow process – its costs, failures, defects, and their genesis – the next step is to design and implement an improved process. What is at issue here is to make improvements that count – that address the biggest gaps in "customer value." These gaps or problems may be "big" because the rate of problems is high, or because the customer is particularly concerned about the problem, or both. It is important to systematically consider the range of problems that could be addressed from both perspectives in selecting areas for improvement.

There is no "silver bullet" for process redesign. Understanding of the root causes of the problems being tackled must be used creatively. Having deployed a new version of the process, the cycle can be started over: monitoring, assessment, and improvement. The one difference is that new monitoring goals can be established to see if the improvements are effective. The result is a continuous process of improvement which places analysis and design effort on high-value problems and adapts to evolving circumstances.

3.5.1 Integrated Support for Assessment and Improvement

The FuseFlow infrastructure will be a particularly suitable substrate for process assessment and improvement. There are many potentially useful tools that would enable and enhance this activity. Two aspects of FuseFlow will make this particularly easy and effective. First, as an integrated system with a natural record or log of its activities, FuseFlow supplies the necessary inputs to process improvement. Second, the products of fuselet workflows are information objects which are uniquely identifiable, and are relatively easy to examine after the fact. Thus, looking for "defects" may involve more fuselets or JBI clients and could be automated in some cases. The monitoring, analysis, and re-design of processes is, itself, a information-centric activity that can benefit from the underlying JBI, and may employ fuselet workflows of its own.

Even external events that may result from workflow outputs will very likely be reflected in information objects due to the anticipated ubiquity of the JBI-supported information spaces.

What sort of tools will support assessment and improvement? Clearly it is a small step from workflow authoring to re-authoring, if the authoring tool is part of an integrated system that includes the workflow management runtime. One design requirement that emerges from this use-case is the ability to find the design artifacts associated with a running (or past) workflow process. Keep in mind that the unique identifier of the FuseFlow specification is used, in part, to identify the fuselets and control objects that execute the workflow. Given a workflow instance, This identifier can be used to find the FuseFlow specification in the library. However, this is not sufficient – as the FuseFlow specification is revised, he workflow process must be associated with the correct version of the specification. FuseFlow specifications do have version numbers in their syntax. A fully integrated system would make these useful by guaranteeing:

- that each version of specification that is instantiated is given an incremented number,
- that each such version is saved in a revision control system, and
- that the executing process is tied to the version number of its original specification.

Similar considerations will arise when considering the relationship between FuseFlow specifications and instances, e.g. keeping track of which parameter values were used for each instance, in each version.

This approach to performance is based primarily on a per-case notion of defects. Thus, the ability to select cases of the workflow based on various criteria will be important. Statistics about the cases (e.g. rates of each kind of defect) and graphs (e.g. defect rates over time) must also be developed. A "database" of cases for each workflow is also envisioned, each case being labeled by its status with respect to performance dimensions, as well as any test results, evaluated transition conditions, and error conditions. Of course this database would be implemented as a collection of information objects. However, it might be adapted so as to be visible within a general-purpose statistical and graphing package, such as the R System [8]. It seems likely that this database could be given a "star-schema" structure, and thus be suitable for OLAP processing. OLAP (*On Line Analytical Processing*) is a data mining technique and associated special-purpose database. OLAP is focused on interactive analysis of very large multi-dimensional datasets. OLAP quickly gives the analyst a wide range of "views" of the data which can emphasize different aspects of it.

Another important capability will be some means of "drilling down" into selected cases of interest. The analyst is envisioned examining the details of some cases, including the information objects that were published and consumed. In addition to the means for displaying the detailed history of a case (e.g. a timeline), this calls for an object browser that can display any information object and can be used to follow object reference "links" that occur in metadata. A *FuseFlow-aware browser* would have navigation that is specially keyed to the various "IDs"

that are part of the FuseFlow system glue (i.e. DesignID, InstanceID, FuseletID, and CaseID). For example, using such a browser, one should be able to navigate from a given object to the workflow that published it (or consumed it), and then back out to other objects published in the same case.

The performance analysis and improvement of each workflow process will be different, and will require some specialized knowledge, as well as creativity. To some extent the domain-specific (and process-specific) knowledge has been accommodated with the performance dimensions of the FuseFlow language. However, the assessment and improvement cycle cannot be completely automated with generic tools and mechanisms. Can any useful generic help for assessment and improvement be provided? The purpose of fuselet workflows is to implement transformations of the information space. Thus problems with a workflow, either in its implementation or with the way its products are used, may be considered to be deficiencies in the information space. All such deficiencies fall into a relatively small set of categories. For each category or class, some suggested methods of diagnosis and remediation can be provided. The initial cut at this generic analysis is an appendix to this report, and is based on the experience to date from this effort, which is somewhat limited.

Six categories of information space problems were identified:

- Missing objects,
- Inaccurate objects,
- Extraneous objects,
- Tardy objects,
- Usability problems, and
- Process too expensive.

These categories are not completely orthogonal. For example, objects might be “missing” because their metadata is inaccurate.

So, what is the role of FuseFlow in this? If the objects which are produced by or consumed by fuselet workflows are important, then the integrated tools for assessment and improvement described in this section may help with diagnosis and remediation. For example, if some objects are “missing” then one strategy is to seek information about the supposed publisher. If the publisher is a TBF in a fuselet workflow, then it may be possible to identify the relevant cases of that workflow, and drill down to see what objects were published. If errors occurred during those cases, that might explain why the objects are missing.

4. Results and Discussion

The principal results of this effort are the FuseFlow language, the design of the FuseFlow system architecture and runtime architecture, and the prototype implementation of parts of the system. These include the

- Central Control Fuselet,
- Task-Bearing Fuselet template,
- ProcessEditor,
- compiler, and
- Monitoring and Control Client.

In addition, a demonstration of using these components was developed – an example workflow authored with the ProcessEditor, compiled with the compiler, then executed and monitored with the Monitoring and Control Client. The fuselets for the tasks of this demonstration are all developed from the Task-Bearing Fuselet template. This demonstration is accompanied by a visualization application that shows the exchange of control objects between fuselets graphically. The prototype components are each sufficiently well developed to support this demonstration, though they are not integrated with one another. Minor hand-editing of the output of the Process Editor is needed for successful compilation.

The language has parts that are well-defined, such as the process definition. It also has parts that are incomplete, such as the facility for formal parameters. Falling somewhere in between are the performance dimensions, where some kinds of performance measures are specifiable, while others are not.

The system architecture for FuseFlow, at a high level, is relatively unchanged from that described in the original proposal. One minor change was to combine the runtime functions (process enactment and monitoring) with workflow instantiation and loading. This makes sense, since these two functions would likely be carried out by the same staff, and the combined program is simpler than two communicating programs would be. Some complexity was envisioned in the instantiation step, due to optimizations when workflows can share fuselets. In the end, this line of research was not pursued, instead ensuring that multiple instances of the same fuselet class can run simultaneously without interference was selected as the higher priority.

"Out-of-band" storage and communications was selected for use for some of the components, such as the FuseFlow library. This provided simpler development of the prototype because some existing code could be reused. However, an alternative that seems preferable in the long run was considered: Place all of the FuseFlow artifacts in the Infosphere as managed information

objects. So, for example, a FuseFlow specification would be stored as an information object. When it is compiled, the result would also be captured in an object. When the workflow instance is loaded, that event would be recorded in an object that would contain the platform-provided fuselet IDs of the loaded fuselets. Continuing in this way, the entire state of the FuseFlow system (and its historical state) would be present in the Infosphere. The difficulty with this scheme is that existing tools, such as iFUSE, do not currently use an Infosphere as their primary storage mechanism, because they are intended to support an off-line mode of operation. Off-line FuseFlow authoring and compiling could be synched-up with a JBI-resident database when the running JBI becomes available. The remaining FuseFlow operations would probably require an on-line JBI in any reasonable implementation.

In the area of process monitoring, assessment, and improvement, a subset of the envisioned process monitoring was implemented. In particular, the "tests" feature of the FuseFlow language does not compile into runtime test results, though this will be straight forward to implement. It became clear that assessing performance and designing improvements was destined to be less automated than had originally been hoped. The analysis of performance data requires insight that can only be expected of human experts. Designing effective improvements will require creativity, at present a trait of humans (only). Tools to support his activity, integrated into the FuseFlow system, could be very helpful.

5. Conclusions

In this effort an integrated fuselet workflow system that works with the Air Force Research Laboratory's JBI platform was designed. Early prototypes of the major components of the system were implemented. No technical showstoppers in the design and implementation work have been encountered thus far. Completion and integration of the FuseFlow system for process design, authoring, enactment, and monitoring will be straight forward. A complete integrated system will allow for a realistic experiment involving real information management problems.

Tools to support performance assessment, diagnosis, and, improvement require further research. This report describes what such tools might be like based on an envisioned process improvement cycle.

The following two observations are based on experience developed by creating example workflows. These are noteworthy, in part, because they are at odds with the expectations at the beginning of the project.

First, workflows, by construction, keep the control-flows between cases well separated. However, it was found, in a number of example workflows, that the desired information transformation required data flow between the cases. That is, a workflow case depended on results from a previous case of the same workflow. When this arises, it negates one of the major anticipated benefits of workflows – that they rationalize the problems of concurrency. Since control cannot flow between cases, there is no way to guarantee that one case completes before the next one begins. These results in forced serialization of processing, at best, and possible race condition or deadlock at worst.

Second, it is expected that the major flows of information would be between fuselets in a workflow, and that the primary input to the workflow would be those objects that trigger a new case. In nearly every example workflow that was created, the information transformation required access to additional objects from the information space, objects not directly associated with the workflow case. A naïve implementation of these workflows would, in some instances, be quite inefficient, since the same object may be required for multiple cases. Also, hand-written code to query for and cache these objects, if it were thread-safe, would need to be carefully written.

Both of these observations suggest that fuselet workflows may not be a panacea for information transformation, that their use would require some care. However, both of these issues can be addressed by enhancements to the workflow abstraction, if necessary. Control structures that allow limited interaction between workflow cases would address the first problem, and access to external "reference objects" can also be usefully abstracted. Thus, workflows with requiring these features would address them in their FuseFlow specification, rather than in hand-written code.

The FuseFlow project has set the stage for a future research and development effort in which a completed system is tested and evaluated in a realistic environment.

6. References

1. Joint Battlespace Infosphere. http://www.rl.af.mil/programs/jbi/

2. Fuselet.org. http://www.fuselet.org/

3. The Workflow Management Coalition. Process Definition Interface – XML Process Definition Language. http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-10-03.pdf

4. The Workflow Management Coalition. http://www.wfmc.org.

5. Organization for the Advancement of Structured Information Standards (OASIS). http://www.oasis-open.org.

6. Carla Marceau and Rob Joyce, "Phase I Final Report – iFUSE: Integrated Fuselet Synthesis Environment", ATC-NY Tech Report TR04-0008, May 19, 2004.

7. David Van Brackle. Fuselet Runtime Execution and Management Environment (FREeME) Final Report. ISX Corporation, Camarillo, CA, December 22, 2004.

8. R Development Core Team. R: A Language and Environment for Statistical Computing. Vienna, Austria, 2006. http://www.R-project.org

9. Hajo A. Reijers, *Design and Control of Workflow Processes*, Springer-Verlag, 2003.

10. Wil van der Aalst and Kees van Hee, *Workflow Management: Models, Methods, and Systems*, MIT Press, 2002.

Appendix A: Notes on Information Space Problems, Diagnosis, and Remediation

A.1 Missing Objects

This class encompasses problems in which objects that "should" have been available to consumers are not. It may be that the objects are really not present. They may have been deleted, or they might never have been published. Alternatively, there may be a problem that prevents the consumer from retrieving the objects. In particular, an incorrect subscription or query may cause this.

Falling into this same category are situations where objects are "desired" (i.e. needed for some purpose) and which are understood to be not currently published. No diagnosis is needed in this case.

A.1.1 Diagnosis – Missing Objects

Institute a search for the missing objects. Best would be to use a different client than the consumer, in order to get a more independent reading on whether the objects exist. If the objects are found, then the problem is with the client.

Examine the supposed publishers of the objects, if these are known. Do these clients exist? Have they run? Did they experience errors? Was the client software updated recently? Was the person operating the software replaced recently? Most importantly, can you *correlate* errors in the producing process with missing objects?

Was the client (consumer) software updated recently? Was the operator of the client software replaced recently? Does the type and version in the client match entries in the MDR?

Does the "user" have permission to get this kind of object? Does some more subtle access control policy prevent access? Check: are denied accesses reported, or are they silently ignored in this client? Perhaps the security audit log will have a record of denials, if that is the problem.

Is this problem intermittent? Was there a time in the past when the objects were reliably present? Is the problem correlated with any other aspect of the situation (e.g. day of the week)? Was there a change in theater operations (e.g. geographic shift)?

Does the publisher rely on other objects? Are these objects reliable? What happens when one of those inputs is not present or is flawed?

Has there been a change in the IM policies regarding object persistence and cleaning up the infosphere?

Based on these considerations, decide between:

- Consumer cannot receive objects because of incorrect type, version, or predicate.
- Consumer cannot receive objects because of security policy. Try to determine whether the policy is in error, or is legitimate.
- Objects have not been published, due to error in the publisher client, incorrect operation/configuration of the publisher, or missing/flawed inputs.
- Objects are still published based on an out-dated profile that no longer corresponds to operations.
- Objects have been deleted from the infosphere, or are only published ephemerally.

The remedies for each of these is typically straight-forward and is not detailed here. As a general strategy, it may be possible to change the producing process so that its operations are more transparent: publish additional objects to form a more complete persistent record of the transaction. Rearrange the process to be more systematic, and to take smaller steps.

A.2 Inaccurate Objects

Problems in this category occur when objects contain metadata that are incorrect, or payloads that are incorrect (or both). Such objects may be received in error, or may be missing in error. They may also convey incorrect information, and poison the correctness of "downstream" processes.

As with missing objects, inaccurate objects can be caused by incorrect producer processes (software and configuration), operator error, or bad or missing inputs.

It is possible that the objects are not really inaccurate. Instead, the consumer (client software or human) is confused or outdated.

A.2.1 Diagnosis – Inaccurate Objects

Correlate inaccurate objects with *failures* and reported errors in the *producer process*. Correlate with other known defects. Look for a plausible causal relationship, or a common cause.

Pick out a cohort of inaccurate objects, and a representative sample of accurate ones. Drill down into the publisher workflow record, to examine the details of how these objects were published. Which objects were inputs? Which other objects were published? Which fuselets or clients were involved? Contrast this with similar investigation of the sample of accurate objects.

Is the inaccuracy a new phenomenon? Is it intermittent? Correlate with changes in the publisher process. Correlate with changes and events in external circumstances (i.e. in theater operations).

Examine the software responsible for the objects: process design and individual fuselets and clients. Has this been recently updated?

Based on these considerations, decide between:

- Inaccurate or missing input objects resulted in publication of inaccurate objects.
- Problem with the software that implements the process results in publication of inaccurate objects – possibly exposed by novel or intermittent circumstances.
- Problem with the way the producer process has been operated or configured.
- Confusion of the consumer or end user.

A.2.2 Remedies – Inaccurate Objects

Direct remedies for these problems are straight-forward.

As with missing objects, enhancing the producer process to be more transparent may be a good first step.

It may also be possible to “label” the objects that are inaccurate, so that they can be avoided. For instance, auxiliary objects can be published that “point to” the good or bad objects. This presupposes that there is some test that can be automatically applied to the objects to determine accuracy at runtime. Such a test could also be built into any consumer process, if it is cheap enough.

A.3 Extraneous Objects

This class consists of problems that cause the consumer to receive additional, irrelevant objects. It can occur because the predicate used is not specific enough. It may also occur because there are extra objects in the infosphere with incorrect metadata that causes them to match the predicate in error.

In either case, these are circumstances that could also fall into classes described above. The diagnosis and remedies are also similar.

A.3.1 Diagnosis – extraneous objects

Example extraneous objects are examined for accuracy (especially of metadata). If inaccuracy is plausibly the cause, then the root cause may be investigated: Correlate extraneous object publication with possible causes. Trace sample extraneous objects back to their origins using the process logs, and investigate possible (process) internal and precursor causes.

If the spurious objects are accurate, then consider whether the predicate used by the consumer is as specific as it could be. Has some change occurred (e.g. software update, software configuration change) that might have changed the predicate? Is there some change in circumstances that might alter the effectiveness of the consumer's predicate?

If the problem lies with the consumer's predicate for object selection, consider whether a tightened predicate will be feasible. Is there sufficient detail in the object metadata to distinguish the relevant objects from the others? If not, then you must address the inadequacy of the metadata.

A.3.2 Remedies – extraneous objects

Generally, if the problem is extraneous objects that are wrong, then the best course is to try to repair their accuracy. This is discussed in Section 2.

If the fault lies with the consumer's predicate, then, if this can be adjusted, that will likely be the best course.

If the object metadata is not sufficiently detailed to allow the necessary selectivity, then there are three possible courses: The objects may be transformed to a type with more detailed metadata. Additional auxiliary objects may be published which augment the metadata, and "point to" the objects that they label. Finally, consumers may be equipped with code that enables them to make their own selection from amongst the objects received (e.g. by examining the payload).

A.4 Tardy Objects

This class of problems comprises those in which objects are not available soon enough to be useful. This can occur for three reasons: the objects are not published early enough, the method of querying for them takes too much time to execute, or the process of retrieving the objects takes too long. This last might occur due to limited bandwidth.

A.4.1 Diagnosis – Tardy objects

For a sample of cases, consider where the time was spent. For each case find the "critical path" in the actual course of execution, and consider the time spent on each of the workflow tasks along this path. Account, also, for the delay which occurs before execution of each task. Analyze, in particular, where the time was spent in the worst cases – suggesting where improvements might be possible.

Consider whether the inputs to the process were timely, and whether that might have delayed the output objects.

Correlate timeliness with other factors that might be causally connected, both internal (to the process) and external.

Compute the nominal delivery time for the objects in question, given the known bandwith and object size. Correlate object size with delay. Consider how many such objects are being delivered.

A.4.2 Remedies – Tardy objects

If the delay occurs in the transport of objects from the IM platform to the consumer, then re-provision the network if possible. If that is not possible, consider how to reduce the size of the objects and/or their number.

If the delay occurs because of query processing (or the equivalent for subscriptions), then this might be speeded up by some combination of:

- Reducing the number of objects in the IOR by cleaning up more aggressively.
- Simplifying the predicate – perhaps selection can be performed more efficiently in the client.
- Pre-selecting and labeling the objects so that a simpler predicate can be used.

A.5 Usability Problems

The consumer (human) may have difficulty using the objects provided because there are too many of them or they are presented poorly. Or, they may be presented in a manner that requires more skill than the user has. In any of these cases, these problems arise because of inadequacy (or inappropriateness) of the application program that is receiving the objects and presenting them. The end user will generally not "see" individual information objects *per se*. Instead, he will be presented with a "picture" composed from information extracted from a set of objects. It is this picture which is either usable or not.

In this light, usability concerns selecting and adapting to alternative, appropriate client programs. From an IM standpoint, this may require transformation to create objects in the right format for a selected application.

A.6 Process Too Expensive

The IM platform is a limited resource in several senses. Processes that create and deliver objects deplete those resources. These costs must be weighed against benefits, and in particular, against the benefit of some competing process' consumption.

This is a vast subject, which will only be touched on here. Costs occur in the infrastructure, in the client program, and with the end user.

In the infrastructure (IM platform, broadly construed) costs include cycles, memory, disk space, bandwidth, and IM staff time. Each of these is influenced by the number and type of objects

published and persisted, the number and complexity of queries and subscriptions, and the number and size of objects delivered (among other factors).

Within the client program bandwidth, cycles, and memory may be limited – essentially constraints imposed by the host platform and network.

Finally, the end user is a limited resource.

A.6.1 Remedies – process too expensive

It is assumed that it has already been determined that some process consumes too much of some resource (i.e. that the produced objects are "too expensive"). Here are a few of the many strategies for reducing costs:

- Shift costs to where they can be borne. For example, using a simpler query, returning more objects, shifts the burden of selection from the IM platform to the client.
- Clean out the IOR to reduce the cost of queries and storage. If possible publish non-persistently.
- Cache and reuse pub and query sequences, whose creation is (in current systems) very expensive.
- Redesign the process to eliminate, speed-up, or de-skill a human task. Address usability problems.
- Slow down the "rate" of processes, or their aggressiveness in publishing. For example, tighten the logic for triggering a new case.
- Delay processes whose outputs are not needed immediately. Redesign these to run as a "batch" at a more convenient time.